Web Performance: Load Testing für HTTP/2 & WebSockets – Framework Evaluierung

Bachelorarbeit

im Studiengang
Medieninformatik

vorgelegt von

Kai-Steffen Reeh
Matr.-Nr.: 27284

am 16. August 2016
an der Hochschule der Medien Stuttgart

Zur Erlangung des akademischen Grades eines Bachelors

Erstprüfer/in:	Prof. Dr. Ansgar Gerlicher
Zweitprüfer/in:	Dr. Daniel Thommes

Kurzfassung

Gegenstand der vorgestellten Arbeit ist die Evaluierung frei erhältlicher Performance Test Frameworks, zusätzlich in Hinblick auf deren HTTP/2 und WebSockets Unterstützung. Zur Einführung wird der allgemeine Aufbau und die Vorgehensweise von Performancetests erörtert. Hierfür werden zwei unterschiedliche Testpläne vorgestellt und miteinander verglichen. Anschließend werden die ausgewählten Frameworks auf ihre Funktionen untersucht. Zuletzt wird beispielhaft eine im Netz frei verfügbare Webanwendung mit den vorgestellten Frameworks getestet und der Ablauf derer jeweils dokumentiert.

Um ein abschließendes Fazit zu formulieren, werden die Frameworks hinsichtlich deren Feature Reichhaltigkeit, Nutzerfreundlichkeit und Aktualität bewertet.

Schlagwörter: Web Performance, Stresstests, HTTP/2, WebSocket, Webanwendung, Frameworks, Performance Programme, Evaluierung

Abstract

Subject of this bachelor thesis is the evaluation of open source performance test frameworks and additionally their support for the HTTP/2 and WebSocket protocols. At first the general structure and execution of a performance test are being described. Two different performance testing plans were chosen to be reviewed and compared. Afterwards the testing frameworks are being tested for their functions. At last an exemplary web application will be tested with the reviewed frameworks. The testing process will be documented.

The frameworks will be rated against each other for their features, usability and up-to-dateness for a last conclusion.

Keywords: Web Performance, Load tests, HTTP/2, WebSocket, Web application, Frameworks, Performance Tools, Evaluation

Eidesstattliche Versicherung

Name:	Reeh	Vorname:	Kai
Matrikel-Nr.:	27284	Studiengang:	Medieninformatik

Hiermit versichere ich, Kai-Steffen Reeh, an Eides statt, dass ich die vorliegende Bachelorarbeit mit dem Titel „Web Performance: Load Testing für HTTP/2 & WebSockets – Framework Evaluierung“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und prüfungsrechtlichen Folgen (§ 26 Abs. 2 Bachelor-SPO bzw. § 19 Abs. 2 Master-SPO der Hochschule der Medien Stuttgart) sowie die strafrechtlichen Folgen (siehe unten) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

Ort, Datum	Unterschrif

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CSV	Comma-separated values
DSL	Domain System Language
FTP	File Transfer Protocol
GUI	Graphical User Interface
HiWi	Wissenschaftliche Hilfskraft
HTML	Hyper Text Markup Language
HTTP(s)	Hypertext Transport Protocol (Secure)
HTTP/2	Hypertext Transport Protocol 2
IMAP	Internet Message Access Protocol
JAX	Java-API for XML-based RPC
JDBC	Java DataBase Connection
JMS	Java Message Service
JSON	JavaScript Object Notation
KO	Fehlgeschlagen
KPI	Key Performance Indicator
LDAP	Lightweight Directory Access Protocol
OK	Erfolgreich
POP3	Post Office Protocol Version 3
REST	Representational State Transfer
RPC	Remote Procedure Call
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SSL	Secure Socket Layer

TCP	Transmission Control Protocol
TPS	Transactions per Second
XML	Extensible Markup Language

1 Einleitung

Webanwendungen werden sehr häufig veröffentlicht, ohne dass sich dabei über ihre Performance Gedanken gemacht wurde. Dadurch kann es zu Komplikationen beim Veröffentlichen der Webanwendungen kommen, wie zum Beispiel eine unvorhergesehen große Anzahl von Besuchern der Webanwendungen. Der Server ist überlastet, die Gründe sind nicht bekannt, es könnte ein SQL-Befehl blockiert sein. Dieser Fehler tritt aber erst bei einer sehr hohen Nutzerzahl auf. Ein mögliches Beispiel ist, dass für die Firma die hinter der Webanwendung steht, hohe Kosten entstehen, während der Server nicht ansprechbar ist. In dieser Arbeit sollen die Fähigkeiten frei erhältlicher Frameworks überprüft werden, in Hinblick für das im Mai 2015 spezifizierte HTTP/2 Protokoll und das in 2011 spezifizierte WebSocket Protokoll. Dabei sind die Durchführungs- und Auswertungsmöglichkeiten der verschiedenen Frameworks gegenüberzustellen.

2 Performance Tests

2.1 Definition Performance im Web

Unter Performance bezeichnet man allgemein die Leistungsfähigkeit. Im Bereich der Webentwicklung spricht man auch davon wie schnell eine Anwendung auf den Nutzer reagiert. Dabei kann es sich um das Abschicken eines Formulars handeln, beispielsweise bei einer Online Banking Seite eine Überweisung für eine offene Rechnung. Oder das Anzeigen der Artikel eines Onlineshops. Die Anwendung benötigt für die Bearbeitung des Auftrags ein bis mehrere Sekunden. Während ein Online Banking Prozess mehrere Sekunden beanspruchen kann, ist es für den Onlineshop fatal, wenn das Laden der Artikel länger als 2 Sekunden dauert. Sollte das Anzeigen beispielsweise länger als 2 Sekunden dauern kann das bereits den Verlust von Kunden bedeuten, die in Zukunft möglicherweise den Shop nicht mehr besuchen werden. Somit wird klar, dass Performance ein relativer Begriff ist und deshalb vom Benutzer und dessen Erwartung abhängig ist.

2.2 Wie Performance gemessen wird

Laut Ian Molyneaux [1] können verschiedene Aspekte der Performance, die sogenannten Key Performance Indicators (KPI), der Anwendung gemessen werden. Diese lassen sich in zwei Typen unterteilen, in Serviceorientiert und effizienzorientiert. Serviceorientiert wird in die Begriffe Verfügbarkeit und Reaktionszeit aufgeteilt, sie messen wie gut die Anwendung ihren Service dem Nutzer zur Verfügung stellt. Effizienzorientiert wird in Durchsatz und Kapazität unterteilt, diese beschreiben wie effizient die Anwendung die Server nutzt. Die Begriffe sind wie folgt definiert [1]:

Verfügbarkeit:

Die Verfügbarkeit zeichnet sich dadurch aus, dass über einen großen Zeitraum die Anwendung verwendbar ist und die Server nicht heruntergefahren sind. Sollten die Server nicht ansprechbar sein, wirkt sich dies unmittelbar auf die Zufriedenheit der Nutzer aus.

Reaktionszeit:

Gemessen in Sekunden. Die Zeit die benötigt wird um eine Anfrage zu bearbeiten. Eine Anfrage kann zum Beispiel das Anzeigen einer Produktliste sein. Die Reaktionszeit bestimmt wie die Anwendung während der Nutzung wahrgenommen wird. Die Zeit sollte möglichst gering ausfallen, damit die Nutzer nicht frustriet sind und die deshalb die Anwendung in Zukunft nicht mehr besuchen.

Durchsatz:

Der Durchsatz gibt die Menge der Aufrufe für einen bestimmten Zeitraum an.

Kapazität:

Die Kapazität gibt an wie viele Nutzer theoretisch uneingeschränkt die Anwendung verwenden können. Zum Beispiel wieviel Bandbreite die Anwendung benötigt oder wie viel Arbeitsspeicher auf dem Server gebraucht werden für eine bestimmte Anzahl Nutzer.

2.3 Performance Wahrnehmung in der realen Welt

Die Performance einer Webanwendung in der realen Welt wird grundsätzlich über die Geschwindigkeit des Ladenvorgangs festgelegt, also die Angabe einer Zeit. Die Frage ist wie lange darf das Laden dauern.

Ian Molyneaux gibt hier beispielhaft an, wie unterschiedliche Zeitfenster in der realen Welt von den Nutzern wahrgenommen werden und welche Anwendungsarten in die genannten Zeitfenster fallen. [2]

Länger als 15 Sekunden:

Für manche Anwendungen oder Nutzer kann diese Zeit zufriedenstellend sein, zum Beispiel aber nicht für Betreiber eines Call-Centers. Sollten solche Verzögerungen auftreten, sollte die Anwendung so gebaut sein, dass andere Tätigkeiten gleichzeitig ausgeführt werden können, sodass man zu einem späteren Zeitpunkt zurückkehren kann um die Antwort zu erhalten.

Länger als 4 Sekunden:

Immer noch zu lange um eine Konversation zu betreiben. Usereingaben würden zu Frustration führen. Jedoch sind 4 bis 15 Sekunden in Ordnung, nachdem eine Transaktion ausgeführt wurde und auf die Bestätigung gewartet wird. Zum Beispiel die Wartezeit nach einer Online Überweisung.

2 bis 4 Sekunden:

Eine Wartezeit von 2 bis 4 Sekunden kann bei der Eingabe von Kreditkartenangaben tolerierbar sein und deren anschließenden Verifizierung, nicht jedoch beim Betrachten von Produkten oder beim Vergleichen von Produkten.

Weniger als 2 Sekunden:

Für Anwendungen bei denen der Nutzer sich viele Informationen merken muss sind kurze Reaktionszeiten sehr wichtig. Je komplexer die Anwendung, desto größer wird der Gebrauch kurzer Zeiten unter 2 Sekunden. Zum Beispiel das Durchblättern der Produkte eines Online Shops.

Weniger als 1 Sekunde:

Denkintensive Anwendungen wie zum Beispiel ein Buch schreiben erfordern sehr kurze Reaktionszeiten, um das Interesse des Nutzers nicht zu verlieren. Ein weiteres Beispiel wäre ein Designer der ein Bild verschieben möchte, dieser benötigt sofortiges Feedback.

Dezisekunden:

Die Reaktion die man erhält, wenn man zum Beispiel auf eine Taste auf der Tastatur drückt und sich ein Charakter auf dem Bildschirm bewegt, oder das Anklicken eines Objektes mit der Maus, muss sofort sichtbar sein. Viele Computerspiele erfordern eine sehr schnelle Reaktion des Nutzers.

Es wird klar, dass eine Reaktionszeit von mindestens weniger als 2 Sekunden für Webanwendungen nötig ist damit der Nutzer nicht das Interesse verliert oder gar Frust aufbaut.

2.4 Performance Test Arten

Folgende Performance Tests werden in dieser Arbeit erwähnt:

- Pipe-Clean-Tests werden ausgeführt, um herauszufinden ob die Anwendung für Performance Tests bereit ist. Es wird überprüft, ob alle Funktionen der Anwendung funktionieren. [3]
- Volume-Tests werden ausgeführt, um sicherzustellen, dass die Anwendung zum Beispiel mit vielen Datenbankeinträgen richtig funktioniert. [4]
- Isolation-Test werden ausgeführt, wenn Fehler entdeckt wurden, aber nicht deren Ursprung. Mit Isolation-Tests können die Quellen der Fehler ausgemacht werden. [5]
- Stresstests werden ausgeführt, um herauszufinden wie das System auf Überlastung reagiert. Dabei ist festzustellen, ob zum Beispiel korrekte Fehlermeldungen ausgegeben werden. [6]
- Soak-Tests werden ausgeführt, um herauszufinden, ob in der Anwendung Speicherlecks existieren, sowie sich im Verlauf des Tests die Reaktionszeiten entwickeln. Für Soak-Tests wird eine große Last über einen langen Zeitraum auf die Anwendung ausgeübt. [7]
- Load-Tests werden ausgeführt, um herauszufinden wie sich die Reaktionszeiten unter verschiedenen Lasteinstellungen, also Nutzeranzahl, verändern. [8]

Es existieren weitere Arten der Performance Tests, welche in dieser Arbeit nicht erwähnt werden, da dies im Rahmen der Arbeit nicht möglich ist. Weitere Testarten lassen sich beispielsweise unter der Webseite Tutorialspoint finden. [9]

3 Erläuterung eines Performance Testplans

Möchte man die Performance seiner Anwendung testen, sollte man sich einen Plan über die Herangehensweise erstellen, um einen Überblick der Anforderungen zu erhalten, als auch eine Aussage über die Dauer des Testens machen zu können, sowie welche Ressourcen benötigt werden. Ian Molyneaux definiert in seinem Buch einen klar gegliederten Plan, der für die Vorgehensweise dieser Arbeit übernommen wird. Das folgende Kapitel 3.1 ist eine Zusammenfassung und Übersetzung, mit übernommenen Überschriften, aus genanntem Buch. [3]

3.1 Performance Testplan

Der Plan lässt sich in sechs Schritte aufteilen

1. Nonfunctional Requirements Capture (NFR)
2. Performance Test Environment Build
3. Use-Case Scripting
4. Performance Test Scenario Build
5. Performance Test Execution
6. Post-Test Analysis and Reporting

3.1.1 Erster Schritt: Nonfunctional Requirements Capture (NFR)

Bevor über das Ausführen der Tests Gedanken gemacht werden können, müssen die Rahmenbedingungen festgelegt werden. Diese sind von mehreren Faktoren abhängig.

- Zuerst ist es wichtig zu wissen bis wann die Performance getestet sein soll, also der zeitliche Rahmen.
- Es sollte bestimmt werden, welche Performance die Anwendung zu erreichen hat.
- Key-Use-Cases müssen identifiziert, dokumentiert und bereit für die Umsetzung sein.

- Benötigte Eingabedaten sowie Datenbankdaten müssen festgelegt und zur Verfügung gestellt werden. Diese Daten sollten in ihrem Inhalt und ihrer Menge möglichst realistisch sein um realitätsnahe Ergebnisse zu erhalten.
- Überprüfen, ob man die nötigen Lizenzen und Resourcen besitzt, um die Tests ausführen zu können.

3.1.2 Zweiter Schritt: Performance Test Environment Build

An diesen Punkt sollte bereits festgelegt sein, welche Hardware, Software und Netzwerkanforderungen für die Testumgebung bestehen. Außerdem sollte die Datenbank, wie bereits erwähnt, mit realistischen Daten gefüllt sein. Und der Server sollte möglichst dem finalen Build entsprechen damit der Test aussagekräftig ist. Für das Aufsetzen der Testumgebung hat Ian Molyneaux folgende Schritte definiert:

1. Genug Zeit einplanen um die Umgebung aufzubauen und zu konfigurieren.
2. Darauf achten, dass man alle Deployment Models einbezieht. Es müssen verschiedene Konfigurationen für LAN und WAN getestet werden.
3. Es ist zu beachten, dass bei vorhandenen externen Links die echten Links verwendet oder die externe Verbindung möglichst realistisch simuliert werden.
4. Genügend Load Injection Kapazitäten[1] zur Verfügung stellen für die vorgesehene Skalierbarkeit.
5. Sicherstellen, dass die Anwendung korrekt in die Testumgebung geladen wurde.
6. Überprüfen, ob man die nötigen Lizenzen besitzt, die in der Anwendung oder unterstützender Software Verwendung finden.
7. Testframework korrekt aufstellen und konfigurieren. Auch hier sollten keine Fehler gemacht werden, die das Testergebnis beeinflussen können.
8. Sicherstellen, dass die KPI Überwachung richtig aufgesetzt und konfiguriert wurde.

3.1.3 Dritter Schritt: Use-Case Scripting

Für jedes Use-Case Skript, das erstellt wurde, sollte folgendes getan werden:

- Die benötigten Daten für eine Session herausfinden.

[1] Als Load Injection Kapazitäten werden die zur Verfügung stehenden Systeme bezeichnet.

- Benötigte Eingabedaten festlegen.
- Festlegen von Checkpoints damit auch die Reaktionszeiten der Abschnitte einzeln innerhalb des Use-Cases gemessen werden können. Außerdem sind diese hilfreich für eine spätere Fehleranalyse.
- Nötige Änderungen der Skripte identifizieren und umsetzen damit der Use-Case richtig ausgeführt werden kann.
- Sicherstellen, dass der Use-Case korrekt wiederholt werden kann, bevor er im Performancetest Anwendung findet.

3.1.4 Vierter Schritt: Performance Test Scenario Build

Bevor man mit dem Testen beginnt, sollte man folgende Fragen beantworten:

- Welche Testarten sollen ausgeführt werden?
- Wie soll die Nachdenkzeit des Users integriert werden?
- Wie viele Load Injector sind nötig?
- Soll der Test für eine festgelegte Zeit ausgeführt werden oder angehalten werden, wenn die zur Verfügung gestellten Daten ausgegangen sind?
- Welche Art des Caching ist auf Nutzerseite zu finden? Ein Nutzer der die Seite bereits besucht hat setzt der Anwendung weniger Last zu, weil gewisse Daten (z.B. CSS-Dateien) nicht erneut heruntergeladen werden müssen.
- Welche Auswirkungen haben die Performance Tools auf die Load Injector Maschinen? Programme können unterschiedlich viel Leistung verlangen wodurch mehr Load Injector Maschinen benötigt werden.

3.1.5 Fünfter Schritt: Performance Test Execution

Ausführen und Beobachten der Tests. Bevor die Tests ausgeführt werden, sollte nochmal eine Generalprobe aller Tests durchgeführt werden, um die Konfiguration zu testen und zu überprüfen ob es keine Probleme mit der Anwendung gibt.

Ian Molyneaux, empfiehlt die Tests in folgender Reihenfolge auszuführen:

1. Führe eine Generalprobe oder Pipe-Clean Test aus als letzter Test für die Test Umgebung. Versichere, dass nichts in der Konfiguration vergessen wurde.

2. Führe einen Volume Test durch. Dabei sollte nach jedem Testdurchlauf die Datenbank auf den Stand vor dem Test zurückgesetzt werden.
3. Werden beim Volume Test Probleme erkannt, sollten diese durch einen Isolationstest genauer untersucht werden. Die entdeckten Probleme sollten den zuständigen Entwicklern weitergegeben werden.
4. Ausführen von Stresstests um herauszufinden wie viel Kapazität insgesamt bei maximaler Last des Servers zur Verfügbar steht. Dadurch lässt sich auch feststellen wie viel Wachstum der Anwendung in Zukunft möglich ist.
5. Führe Langzeit Tests (Soak Tests) aus, um mögliche Speicherlecks zu entdecken. Möglicherweise ist dieser Test nicht durchführbar, jedoch ist es zu empfehlen einen Soak-Test auszuführen.
6. Führe Tests, die keinen Bezug zur Performance haben aus. Zum Beispiel verschiedene Lastverteilungen überprüfen.

3.1.6 Sechster Schritt: Post-Test Analysis and Reporting

Der letzte Schritt enthält das Prüfen und Auswerten der gesammelten Daten und mögliches erneutes Testen nach dem gefundene Fehler in der Anwendung korrigiert wurden. Folgendes sollte unternommen werden:

- Daten sammeln. Für alle Daten ein Backup erstellen, um auch später mögliche Probleme zu entdecken, die übersehen wurden.
- Feststellen, ob die Performanceziele erreicht wurden oder nicht.
- Dokumentieren der Ergebnisse.
- Die Ergebnisse können für spätere end-user experience Überwachung als Messbasis verwendet werden.

3.2 Performance Testing nach Oracle

Dieses Kapitel ist eine Zusammenfassung und Übersetzung der Onlinedokumentation von Oracle [11], die sich auch dem Thema Performance Testing widmet und einen Plan aufstellt der dem von Ian Molyneaux ähnelt.

3.2.1 Planung der Lasttests nach Oracle

Auch Oracle beginnt mit einer Liste theoretischer Fragen, die man beantworten sollte bevor man sich an das Testen macht. Oracle unterteilt das Performance Testen in drei Schritte ein [11]:

1. Ziele und Anforderungen für den Skalierbarkeitstest festlegen.
2. Methodik: Der Prozess um Skalierbarkeit und Performance, während aller Phasen der Entwicklung der Anwendung, zu gewährleisten.
3. Planung und Ausführung der Tests.

3.2.2 Ziele und Anforderungen

3.2.2.1 Ziele von Skalierbarkeitstests

1. Ermitteln der maximalen Nutzer der Anwendung:
    - Das Nutzerlimit ist die maximale Anzahl von Nutzern, die das System unterstützt ohne, dass sich für die Nutzer eine bemerkbare Veränderung der Reaktionszeit ergibt.
    - Das Limit sollte höher sein als die Anzahl gleichzeitiger Nutzer, die das System unterstützen soll.
2. Ermitteln des clientseitigen Performanceabbaus und Nutzererfahrung unter Last:
    - Können die Nutzer die Anwendung rechtzeitig erreichen?
    - Ist es möglich die Aktionen der Nutzer in einer akzeptablen Zeit auszuführen?
    - Wie wird die Anwendung durch gleichzeitige Nutzer, Transaktionen und Nutzen beeinträchtigt?
    - Verhält sich die Anwendung unter schwerer Last richtig? Werden Seiten korrekt oder nur teilweise geladen? Stürzen Komponenten ab?

- Wie hoch ist die Fehlerrate, die die Nutzer wahrnehmen und ist diese akzeptabel?

3. Ermitteln der serverseitigen Robustheit und Performanceabbau:
    - Stürzt der Web-Server unter hoher Last ab?
    - Stürzt der Anwendungsserver unter hoher Last ab?
    - Stürzen andere Middle-Tier-Server ab?
    - Stürzt der Datenbankserver unter hoher Last ab?
    - Benötigt das System Lastverteilung? Wenn diese bereits existiert, arbeitet sie fehlerfrei?
    - Kann die bestehende Architektur besser konfiguriert werden?
    - Sollte die Hardware für bessere Performance geändert werden?
    - Gibt es Deadlocks im System?

3.2.2.2 Phasen der Skalierbarkeit Tests

Folgende Phasen der Skalierbarkeit sind [12]:

Architektur Validierung – Überprüfen der Architektur in frühen Entwicklungsphasen. Möglichst nachdem ein Prototyp der Anwendung erstellt wurde mit allen existierenden Transaktionen. So lässt sich früh bestimmen ob die Anwendung realisierbar ist.

Performance Benchmarking (Leistungsvergleich) - Testen einer ersten Version der Anwendung für alle Transaktionen und Rücksprache mit den Entwicklern, ob die Skalierbarkeit basierend auf den festgelegten Anforderungen weiter angepasst werden muss oder nicht.

Performance Regression (**Leistungsabfall)** – Nachdem Meilensteine erreicht oder die Architektur angepasst wurden, wird die Anwendung mit den bereits erstellten Benchmarks verglichen. Es ist darauf zu achten, dass die Änderungen keinen negativen Einfluss auf die Skalierbarkeit der Anwendung haben.

Acceptance and Scalability Fine Tuning – Finale Phase vor Veröffentlichung der Anwendung. Software, Hardware und Load Balancing Komponenten werden zusammengefügt und ein weiteres Mal mit verschiedenen Szenarien die Skalierbarkeit getestet.

24x7 Performance Monitoring – Nachdem die Anwendung veröffentlicht wurde, ist es notwendig sie weiter unter realer Last zu beobachten, um eventuelle Crashs oder Ver-

schlechterungen der Reaktionszeit zu entdecken. Des Weiteren können die Nutzerdaten für zukünftige Tests hilfreich sein.

3.2.3 Kriterien für genaue Skalierbarkeitstests

Damit der Performancetest möglichst der Realität entspricht muss das ausgewählte Tool gewisse Fähigkeiten beherrschen [13]:

- Last auf allen Schichten einer Mehr-Schichten-Anwendung erzeugen.
- Erlauben von gemischten Simulationen verschiedener Benutzergruppen.
- Emulieren verschiedener Request Muster von beliebten Browsern.
- Validierung der Server Antworten unter Stress auf korrekte Ergebnisse.
- Erlaubt unkomplizierte Bearbeitung der Skripte falls Änderungen in der Anwendung gemacht wurden, um diese erneut testen zu können.

Weitere wichtige Kriterien, die das Tool beherrschen sollte:

- Dynamische Einbindung neuer Nutzer während der Test noch läuft.
- Möglichkeit zur Überwachung eines einzelnen Testnutzers während des Testablaufs.
- Real-Time Graphen um während dem Testen die Skalierbarkeit zu beobachten.
- Möglichkeit Tests verteilt mit mehreren Maschinen ausführen zu können, über LAN oder WAN.
- Erlaubt das Einbinden verschiedener Nutzer Nachdenkzeiten (zufällig, festgelegte, keine).
- Messen der Response Zeiten der gesamten Transaktion, einzelner Objekte und einzelner Bilder oder Dateien.
- Testen verschiedener Caching Verhalten.
- Einbindung von Eingabedaten.
- Unterschiedliche Ablaufpläne zur Erzeugung verschiedener Szenarien (starten, stoppen, anlaufen).
- Stellt Berichte zur Verfügung für spätere Auswertung und Vergleiche mit zuvor erstellten Berichten.

3.2.4 Festlegen von weiteren benötigten Tools

Bevor mit dem Testen angefangen werden kann, müssen die verschiedenen Tools auf die benötigten Fähigkeiten überprüft und beurteilt werden, ob diese zum Einsatz kommen sollen. Im Falle des Oracle Testplans müssen die Oracle Programme betrachtet werden [14]:

Oracle OpenScript - Wird verwendet um die Use-Cases zu scripten. Die Skripte stellen den Ablauf der Interaktion des Benutzers mit der Anwendung dar. Diese müssen mit Änderung der Anwendung ebenfalls angepasst werden.

Oracle Load Testing - Erstellt virtuelle Nutzerprofile und Szenarien und führt die Load Tests basierend auf den Skripten aus. Mehrere Nutzer werden simultan ausgeführt, um die Skalierbarkeit der Anwendung zu testen. Ermöglicht die Darstellung der Ergebnisse in Graphen für eine bessere Auswertung.

Oracle Load Testing ServerStats - Überwacht den Server in Echtzeit, um herauszufinden wie der Server auf den Test reagiert.

Other System Monitoring Tools - Ermittlung zusätzlich benötigter Software.

Logging Tools - Festlegen der Software die verwendet werden soll zum Aufzeichnen von Fehlern.

3.2.5 Ermitteln nötiger Hardware

Für das Ausführen der Tests muss überlegt werden von welcher Hardware aus die Performance Tests ausgeführt werden. Damit man die Last von vielen gleichzeitigen Nutzern simulieren kann, muss man folgendes berücksichtigen [15]:

Load Distribution Capability - Ermöglicht das Test Tool Last von mehreren Maschinen aus zu generieren und von einem Punkt aus zu kontrollieren?

Operating System - Welches Betriebssystem eignet sich am besten für die Lasttests

Processor - Welche Prozessortypen sind für die Master und virtuellen Agenten nötig?

Memory - Wie viel Speicher ist für die Master und virtuellen Agenten nötig?

Allgemein gilt:

- Stabile und gut skalierbare Systeme sind am besten geeignet.

- Sollte die CPU Auslastung zwischen 70% und 80% liegen oder sollte der Speicherverbrauch höher als 85% sein, könnten die Ergebnisse nicht verwertbar sein.
- Man sollte erwägen den Lastverwalter getrennt von den virtuellen Nutzern laufen zu lassen, da dieser mehr Leistung benötigt und die virtuellen Nutzer auf mehreren getrennten Maschinen laufen können.
- Die Anzahl der virtuellen Nutzer lässt sich ungefähr bestimmen, da ein virtueller Nutzer der als Thread in einem Prozess läuft benötigt ca. 300-500KB Speicher. Laufen diese als separate Prozesse in einem Prozess ca. 1024-2048MB Speicher.
- Normalerweise werden von den Test Tool Entwicklern die empfohlenen Hardware-Spezifikationen angegeben.

3.2.6 Wer soll die Performance Tests durchführen?

Die Performance Tests sollten Personengruppen durchführen, welche aktiv an Ihnen beteiligt sind [16]:

Development Engineers and Architecture Groups - Entwerfen Architektur-Validierungstests und stimmen die Anwendung optimal ab.

Quality Assurance Organizations - Entwerfen und Ausführen von Unit-Tests um die richtige Ausführung der Anwendung sicherzustellen.

Integration and Acceptance Organizations – Versichern, dass alle Schichten korrekt miteinander arbeiten.

Monitoring and Operations Groups - Entwerfen Überwachungstests, um sicherzustellen, dass die Anwendung nach Veröffentlichung 24 Stunden in der Woche zur Verfügung steht und dessen Performance sich nicht unter Last und regelmäßiger Nutzung verschlechtert.

3.2.7 Was beim Testen verhindert werden sollte

Um korrekte Ergebnisse zu erhalten, sollte man folgendes nicht tun [17]:

- Die Performance einer Anwendung testen, die noch entwickelt wird und sich stetig verändert.
- Anwendungen testen, deren Funktionen noch nicht getestet worden sind.

- Teile der Anwendung testen und dessen Ergebnisse auf die gesamte Anwendung beziehen.
- Mit wenigen Nutzern testen und das Ergebnis für viele Nutzer hochrechnen.

3.2.8 Ausführen der Skalierbarkeitstests

Generelles Verfahren für die Ausführung der Skalierbarkeitstests [18]:

- Kriterien für die Skalierbarkeit bestimmen.
- Benötigte Tools bestimmen.
- Konfigurieren der Hardware und Umgebung, die für die Tests benötigt wird bestimmen.
- Planen der Skalierungstests.
- Planen der Test-Szenarien.
- Erstellen von Test-Skripten.
- Erstellen der Test-Szenarien.
- Ausführen der Tests.
- Beurteilung der Ergebnisse anhand der Kriterien.
- Erstellen des Berichts.

Die einzelnen Schritte werden nachfolgend erklärt.

3.2.8.1 Entwerfen des Prozesses

Sobald die benötigten Anforderungen für die Last-Tests definiert sind, sollte der Ablauf definiert werden. Einige Fragen und Probleme sollten vorher beachtet werden. [19]

Benötigte Anwendungen – Welche Anwendung sollen verwendet werden für das Performancetesten?

Ablaufplan - Wann werden die Tests ausgeführt? Welche Meilensteine müssen erreicht werden?

Personal - Wer führt die Analyse, Planung, Test-Entwicklung, Test-Ausführung und Evaluation aus? Werden Drittanbieter (Frameworkanbieter, Internet Service Provider, Testlabor) benötigt?

Ort - Wo werden die Tests ausgeführt? Intern, extern in einem Labor oder bei einem Internet Service Provider?

Testumgebung - Auf welcher Software/Hardware werden die Lasttests ausgeführt? Beim Auswählen der Testumgebung sollten häufige Fehler vermieden und beachtet werden:

- **Stabilität der Anwendung** - Die Anwendung sollte während der Tests nicht geändert werden.
- **Einsatzumgebung** - Die Testumgebung sollte möglichst der realen Umgebung gleichen.
- **Aufnahmeumgebung** - Bevor die Anwendung ausgeliefert wird, sollte ein weiterer Test auf einer Umgebung, die exakt der realen Umgebung gleicht, ausgeführt werden.

Hardware Zuordnung - Ist die nötige Hardware zugeordnet und einsatzbereit? Test Framework Entwickler können bei der Wahl der Hardware basierend auf folgenden Informationen helfen:

- Anzahl virtueller Nutzer
- Maximal erlaubte Reaktionszeit eines Vorgangs
- Maximal erlaubte Verzögerungsdauer zwischen zwei Vorgängen

3.2.8.2 Festlegen der Kriterien

Bevor die Anwendung ausgeliefert werden kann, muss festgelegt werden unter welchen Kriterien dies geschehen soll. Beim Festlegen der Kriterien sollte folgendes spezifiziert werden [20]:

Die zu simulierende Last - Wie viele virtuelle Nutzer müssen simuliert werden? Die Angabe beinhaltet wie viele Nutzer gleichzeitig auf dem Server sind.

Anzahl der zu simulierenden Vorgänge - Wie viele Vorgänge müssen für den Test simuliert werden?

Welche Vorgangstypen sind zu simulieren - Beispiele: Auslesen des Kontostandes, eine Überweisung erstellen, Kontodetails ansehen, etc.)

Kriterien für jeden einzelnen Vorgang - folgendes sollte definiert werden:

- Akzeptable Reaktionszeiten unter verschiedenen Anwendungsauslastungen.
- Akzeptable Fehlerrate unter Last festlegen.

- Userkategorien: Erstbesucher oder wiederkehrende Besucher. Erstbesucher verursachen mehr Last, weil diese alle Daten vom Server benötigen und keine Daten der Seite im Cache besitzen.
- Überprüfen ob der Test SSL und HTTP oder nur eines der beiden benötigt.
- Testen von verschiedenen Browsern.
- Festlegen ob Nutzer-Denkzeiten eingebaut werden sollen oder nicht. Sollen Tests ohne Denkzeiten zur Überprüfung der Stabilität ausgeführt werden oder zufällig generierte Denkzeiten verwendet werden, um verschiedene Nutzer zu simulieren?
- Festlegen der Wartezeiten, die zwischen Vorgängen eingebaut werden sollen.
- Testen der Last mit und ohne Laden von Bildern.

Overall Transactions-Per-Second Throughput Required - Wie viele Vorgänge müssen innerhalb einer Sekunde im Test ausgeführt werden?

Error Handling – Was muss getan werden, wenn ein Fehler auftritt? Muss die Anwendung während dem Auftreten gestoppt werden oder werden Fehler aufgezeichnet?

Vorgangstypen und Performance aufzeichnen – Welche Typen der Vorgänge und Performance müssen für die verschiedenen Skripte aufgezeichnet werden?

3.2.8.3 Den Skalierungstest planen

Vor der Entwicklung der Testskripte sollte ein detaillierter Testplan erstellt werden. Für jeden Vorgang, der im Test ausgeführt wird, müssen folgende Informationen geplant und definiert werden [21]:

Steps for Scripts, Schritte für die Skripte – Jedes Skript sollte eine detaillierte Sequenz exakter einzelner Aktionen, die ausgeführt werden sollen, enthalten.

Run-Time Data, Laufzeitdaten – Der Testplan sollte spezifizieren, welche Daten während der Ausführung der Skripts benötigt werden, um mit der Anwendung interagieren zu können (Logins, Passwörter).

Data Driven Tests, Daten abhängige Tests - Angaben der benötigten Eingabedaten für die Interaktion mit der Anwendung, zum Beispiel Daten aus Datenbanken oder erstellen fiktiver Daten.

3.2.8.4 Test Szenarien planen

Zusätzlich zu den Aktionsdetails der Skripte muss spezifiziert werden, welche verschiedenen Nutzergruppen für die Testszenarios benötigt werden. Für jedes Testszenario müssen folgende Informationen festgelegt werden [22]:

Nutzertyp – Nutzer, die die Anwendung das erste Mal besuchen oder bereits besucht haben.

Auszuführende Aktionen – Welche Aktionen werden die Nutzer ausführen und in welcher Reihenfolge.

Nutzeranzahl – Anzahl virtueller Nutzer mit einem bestimmten Nutzerprofil welche die Anwendung in einem angegebenen Zeitfenster verwenden.

Das System – Welche PC Systeme werden verwendet um Nutzerlast zu generieren.

Die Browser – Welche Browser nutzen die verschiedenen Nutzerprofile.

Durchführungstempo - Wie schnell interagieren die Nutzergruppen mit der Anwendung. Diese können unterschiedlich definiert werden. Aufgenommene Denkzeiten, einen Raum von verschiedenen Denkzeiten, oder zufällig generierte Nachdenkzeiten.

Verzögerungen - Welche Verzögerungen existieren zwischen Vorgängen, falls welche existieren.

Bilder – Es sollten verschiedene Nutzergruppen definiert werden, welche mit oder ohne Bilder laufen, um die verschiedenen Lasten, die die Bilder auslösen können, zu testen.

3.2.8.5 Erstellen und verifizieren von Testsskripten

Nach dem Planen der Tests müssen die Skripte, basierend auf den folgenden festgelegten Szenarien, im Testplan erstellt werden [23]:

- Auszuführende Nutzeraktion
- Zeitnehmer
- Tests ausführen
- Datenquellen

Für jedes Skript muss sichergestellt werden, dass das Skript die Aktionen richtig ausführt und die zu erwartenden Ergebnisse liefert.

3.2.8.6 Erstellen und verifizieren Testszenarien

Sobald die individuellen Skripte erstellt und überprüft wurden, können diese zu Testszenarios zusammengefügt und verifiziert werden. [24]

Überprüfung der Skripte mit mehreren virtuellen Nutzern – Bevor die Skripte zu Testszenarios zusammengefügt werden können, muss überprüft werden, ob ein einzelnes Skript mit mehreren virtuellen Nutzern funktioniert und den Kriterien entspricht.

Verifizierung der verteilten Testausführung auf mehreren Maschinen - Falls für die Testausführung mehrere Maschinen geplant sind, sollte überprüft werden, ob das Testframework die nötigen Fähigkeiten für eine verteilte Ausführung besitzt.

Verifizierung der realen Szenarien mit jeder Nutzergruppen - Bevor ein Test mit mehreren virtuellen Nutzern aus allen Nutzergruppen ausgeführt wird, muss überprüft werden, ob die Ausführung mit nur einem Nutzer aus einer Nutzergruppe die zu erwartenden Ergebnisse liefert. Dies muss für alle Nutzergruppen überprüft werden.

Erstellung reale Szenarien – Die einzelnen Szenarien sollten folgende Informationen enthalten**:**

- Nutzertypen
- Denkzeiten
- Auszuführende Navigation/Aktion
- Verzögerung zwischen Aktionen
- Anzahl der Nutzer verschiedener Nutzergruppen
- Mit oder ohne Bilder
- Verwendetes System zur Lastgenerierung
- Fehlerlog Einstellungen
- Browser Emulation

3.2.8.7 Ausführen der Tests

Wenn alle Testszenarios überprüft wurden, kann mit der Ausführung derer mit mehreren virtuellen Nutzern beginnen. Es kann nun davon ausgegangen werden, dass die Ergebnisse valide sind. [25]

Ausführung einfacher Tests aus, um Skalierbarkeit zu testen - Führe die Tests mit minimalen Nutzerzahlen aus, um sicherzugehen, dass die Anwendung richtig skaliert:

- Ausführen einzelner Transaktionen beginnend mit zehn Nutzern steigern bis zu 25 – 50 Nutzern.
- Führe eine Kombination verschiedener Vorgänge aus, beginnend mit fünf Nutzern Skalierung bis zu 25 Nutzern.

Sollten die beiden genannten Tests erfolgreich sein, können die vollständigen Skalierbarkeitstests mit der gesamten Anzahl benötigter virtueller Nutzer ausgeführt werden.

Ausführung der realen Szenarien - Führe alle realen Szenarien aus, die in den vorherigen Schritten entworfen wurden:

- Erhöhe die Anzahl der Nutzer für die Szenarien um die benötigte Höhe.
- Überwache den Test auf mögliche Fehler.

Erneute Ausführung der Szenarien mit realen Nutzern - Während ein Test läuft sollte ein realer Nutzer die Anwendung simultan verwenden, um über die Performance Aussagen zu treffen:

- Beobachte mögliche Verschlechterung der Reaktionszeiten für den realen Nutzer.
- Beobachte, ob Fehler im Browser zurückgegeben werden.

3.2.8.8 Evaluierung der Ergebnisse

Für jedes Testszenario müssen die Performancedaten ausgewertet werden und mit den zu Beginn festgelegten Kriterien verglichen werden [26]:

- Reaktionszeiten verschiedener Nutzergruppen und Nutzerzahlen.
- System Durchsatz bei verschiedenen Nutzeranzahlen.
- Aufgetretene Fehler überprüfen.

3.2.8.9 Analysebericht

Dokumentieren der Performance durch Erstellen von Berichten die nötig sind für die Akzeptanz der Anwendung und für deren Auslieferung. Nachfolgend einige Beispiele zu Gutachtentypen, die erstellt werden können [27]:

- Performance vs. Zeit
- Statistik vs. Zeit
- Nutzer vs. Zeit

- Fehler vs. Nutzer
- Statistik vs. Nutzer
- Fehler vs. Zeit
- Sonstige Fehlergutachten die möglicherweise später von Relevanz sind für Entwickler, um mögliche Fehler zu beheben.

3.3 Vergleich Oracle und Ian Molyneaux

Obwohl, dass es keinen definierten Standard für die Vorgehensweise des Performance Testing gibt, sind sich Oracle und Ian Molyneaux im Ablauf der Ausführung der Performance Tests einig. Als Erstes muss sich Gedanken gemacht werden, welche Performance erreicht werden möchte, darauf folgend welche Tools und Systeme zur Verfügung stehen, um die Testpläne ausführen zu können. Der nächste Punkt ist das Entwerfen der Szenarien und Skripte, sowie schließlich die Umsetzung dieser. Zuletzt werden die Szenarien ausgeführt, die Ergebnisse erstellt und ausgewertet. In dieser Arbeit wird der Performance Testplan nach Ian Molyneaux für die Ausführung der Tests verwendet.

4 Testing Frameworks

Die Kriterien der zu testenden Frameworks in dieser Arbeit sind wie folgt:

- Sie müssen Open Source sein.

Folgende Open Source Frameworks wurden anhand dem Bachelorarbeitaushang ausgewählt:

- Gatling
- JMeter mit Blazemeter Plugin
- The Grinder

Diese werden in den folgenden Abschnitten vorgestellt und auf deren Eigenschaften und Besonderheiten sowie Vorzüge eingegangen.

4.1 Gatling.io

Nachfolgend in dieser Arbeit Gatling genannt, wurde mit dem Gedanken entwickelt, dass es leicht zu nutzen und zu pflegen ist, sowie eine hohe Performance besitzt. Gatling setzt auf eine asynchrone Implementierung solange wie auch das darunter liegende Protokoll nicht blockierend ist [28].

Schlüsselfunktionen von Gatling sind [29]:

1. HTTP Recorder
2. Scala als DSL (Domain System Language) für die Test Entwicklung. Scala ist eine Programmiersprache, die auf einer Java Virtual Machine läuft.
3. Hohe Last möglich durch nicht blockierendes asynchrones Verfahren
4. Volle Unterstützung für HTTP(S) so wie WebSocket, JDBC und JMS Lasttests
5. Unterschiedliche Möglichkeiten zur Integration von Testdaten (CSV, JSON, Redis)
6. Möglichkeit zur Validierung und Überprüfung der Erwartungen
7. Umfassende und informative Ergebnisberichte
8. Fähigkeit zur Realtime Monitoring

4.1.1 Die GUI

Gatling besitzt nur für die Erstellung der Skripte eine GUI, für die Ausführung der Skripte wird ein Konsolenkommando verwendet, dazu mehr in Kapitel 4.1.5. Auf die GUI für die Erstellung der Skripte, wird im nächsten Kapitel näher eingegangen.

4.1.2 Erstellung der Use-Case Skripte

Die Tests können mit Hilfe der Skript-API oder direkt in Scala geschrieben werden. Die Skripte sind lesbar und einfach instand zu halten.

4.1.2.1 Script Recorder

In der Vorstellung der Frameworks wurde bereits erwähnt, dass sich die Use-Case-Skripte bequem mit einem Recorder aufnehmen lassen, indem man mit der Anwendung interagiert. [30] Dadurch lassen sich echte Reaktionszeiten simulieren. Ist ein Skript erstellt, so lässt sich dieses erweitern in dem man die erstellte „scala“ Datei anpasst. Dazu bietet Gatling eine Online Dokumentation zum Nachschlagen der existierenden Befehle an. Die Oberflächen des Recorders sehen wie folgt aus:

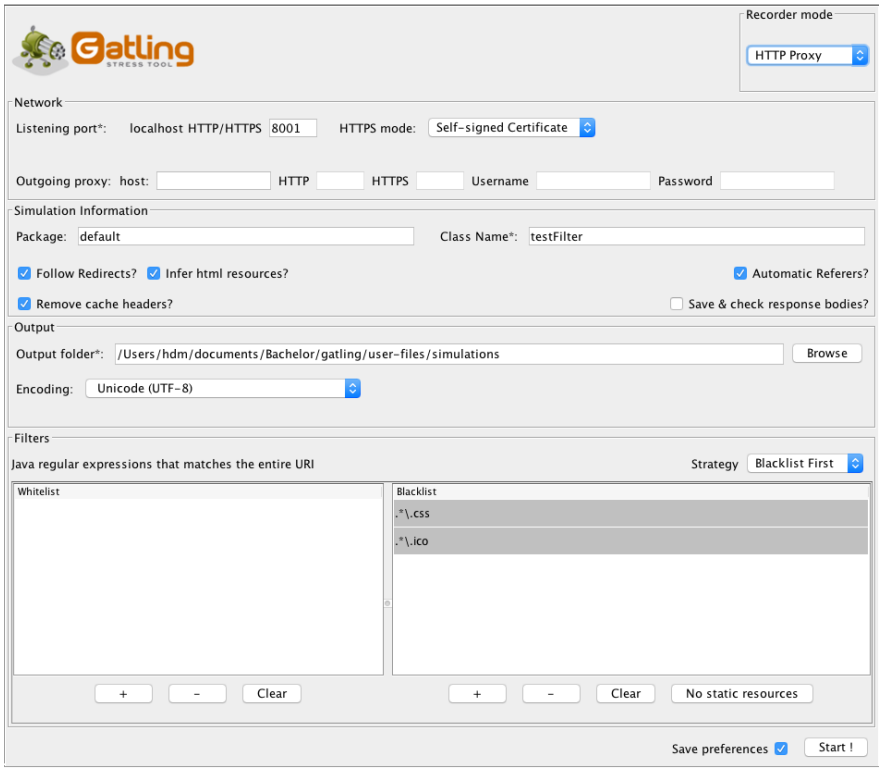

Abbildung 1: Recorder GUI

In Abbildung 1 sieht man die Konfiguration der Recorder GUI. Hier kann der Proxy eingestellt, Informationen über die Simulation angegeben, der Ausgabeort und Filter eingestellt werden.

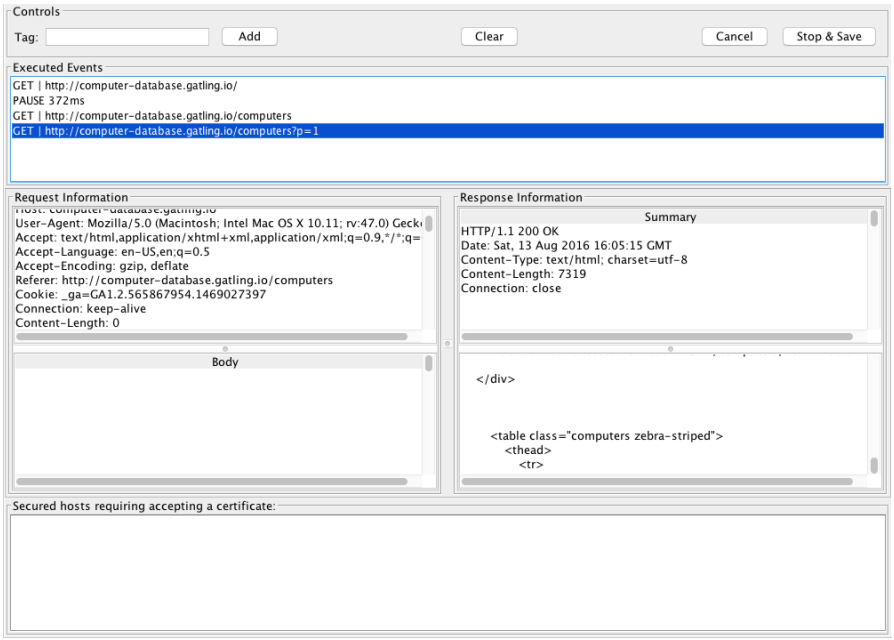

Abbildung 2: Recorder GUI #2

In Abbildung 2 kann betrachtet werden, welche Aktionen der Recorder aufgenommen hat. Wählt man eine der Aktionen aus werden weitere Details über diese angezeigt. Ist die Ausführung des Szenarios fertig wird dieses gespeichert durch anklicken des „Stop & Save" Knopf. Basierend auf die Use-Cases in Kapitel 5.1, wurde folgendes Skript erstellt:

```
class RecordedSimulation extends Simulation {

   val httpProtocol = http
      .baseURL("http://computer-database.gatling.io")
      .inferHtmlResources()

.acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,*
/*;q=0.8")
      .acceptEncodingHeader("gzip, deflate")
      .acceptLanguageHeader("en-US,en;q=0.5")
      .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11;
rv:47.0) Gecko/20100101 Firefox/47.0")

   val headers_1 = Map("Accept" -> "*/*")

   val scn = scenario("RecordedSimulation")
      .exec(http("request_0")
         .get("/")
         .resources(http("request_1")
         .get("/favicon.ico")
         .headers(headers_1)
```

```
          .check(status.is(404)),
             http("request_2")
          .get("/favicon.ico")
          .check(status.is(404))))
       .pause(8)
       .exec(http("request_3")
          .get("/computers?f=macbook"))
       .pause(9)
       .exec(http("request_4")
          .get("/computers/6"))
       .pause(17)
       .exec(http("request_5")
          .get("/computers"))
       .pause(5)
       .exec(http("request_6")
          .get("/computers?p=1"))
       .pause(2)
       .exec(http("request_7")
          .get("/computers?p=2"))
       .pause(2)
       .exec(http("request_8")
          .get("/computers?p=3"))
       .pause(1)
       .exec(http("request_9")
          .get("/computers?p=4"))
       .pause(12)
       .exec(http("request_10")
          .get("/computers/new"))
       .pause(19)
       .exec(http("request_11")
          .post("/computers")
          .formParam("name", "Test")
          .formParam("introduced", "2016-01-01")
          .formParam("discontinued", "")
          .formParam("company", "1"))

   setUp(scn.inject(atOnceUsers(1))).protocols(httpProtocol)
}
```

Die Eingabedaten werden automatisch mit dem Recorder aufgenommen. Aus diesem Grund ist das Skript mit dem Gatling Konsolenprogramm nach der Aufnahme direkt ausführbar.

4.1.2.2 Isolierte Prozesse

Ein wichtiger Bestandteil eines Performancetests ist die Möglichkeit nur einzelne Prozesse ausführen zu können, um entdeckte Fehler weiter untersuchen zu können. Bei Gatling geschieht das, indem die einzelnen Aktionsschritte in einem Objekt gespeichert werden. Die Objekte können in neuen Dateien gespeichert oder direkt in der Simulationsdatei geschrieben werden. [31]

Nachfolgend ist zu sehen wie die Erstellung isolierter Prozesse mit Objekten aussehen könnte [31]:

```
object Search {

  val search = exec(http("Home")
    .get("/"))
    .pause(7)
    .exec(http("Search")
    .get("/computers?f=macbook"))
    .pause(2)
    .exec(http("Select")
    .get("/computers/6"))
    .pause(3)
}

object Browse {

  val browse = exec(http("request_7")
      .get("/computers?p=2"))
      .pause(2)
      .exec(http("request_8")
      .get("/computers?p=3"))
      .pause(1)
      .exec(http("request_9")
      .get("/computers?p=4"))
}

object Edit {

  val edit = .exec(http("request_10")
      .get("/computers/new"))
      .pause(19)
      .exec(http("request_11")
         .post("/computers")
         .formParam("name", "Test")
         .formParam("introduced", "2016-01-01")
         .formParam("discontinued", "")
         .formParam("company", "1"))
}

val scn = scenario("Scenario Name").exec(Search.search, Browse.browse,
Edit.edit)
```

4.1.2.3 Konfigurieren der virtuellen Nutzer

Ein unerlässlicher Bestandteil, der Performance Tests überhaupt möglich macht, ist die Erstellung mehrerer virtuellen Nutzer. Die Anzahl der Nutzer kann in der Methode im Skript geändert werden `setUp(scn.inject(atOnceUsers(1))).protocols(httpProtocol)` in dem man die Zahl der Methode erhöht `atOnceUsers(1)`.[32]

So lassen sich Stresstests simulieren, indem man möglichst viele Nutzer einstellt. Möchte man jedoch eine realistische Nutzerinteraktion generieren sollen die User Stück für Stück auf den Server gelassen werden. Um das zu erreichen muss folgendes im Skript geschrieben werden:

```
setUp(
  users.inject(rampUsers(10) over (10 seconds)),
  admins.inject(rampUsers(2) over (10 seconds))
).protocols(httpConf)
```

In der Dokumentation finden sich weitere Möglichkeiten wie die Nutzer injiziert werden können [33]:

```
setUp(
  scn.inject(
//  Tu nichts für X Zeitangabe
    nothingFor(4 seconds),
// Injitziere X Nutzer auf einmal
    atOnceUsers(10),
// Injitziere X Nutzer über X Zeitangabe
    rampUsers(10) over(5 seconds),
// Konstante X Nutzer über X Zeitangabe
    constantUsersPerSec(20) during(15 seconds),
// Konstante X Nutzer über X Zeitangabe, in zufälligen Intervallen
    constantUsersPerSec(20) during(15 seconds) randomized,
// Injiziert X bis Y Nutzer in Z Zeitangabe
    rampUsersPerSec(10) to 20 during(10 minutes),
// Injiziert X bis Y Nutzer in Z Zeitangabe, in zufälligen Intervallen
    rampUsersPerSec(10) to 20 during(10 minutes) randomized,
// Injiziere Y Nutzer, getrennt von Injetzierung Z Nutzer, bis X
Nutzer erreich sind
    splitUsers(1000) into(rampUsers(10) over(10 seconds))
separatedBy(10 seconds),
// Injiziere Y Nutzer, getrennt von Z Pause, bis X Nutzer erreich sind
    splitUsers(1000) into(rampUsers(10) over(10 seconds)) separatedBy
atOnceUsers(30),
// Injiziert X Nutzer über Y Zeitangabe, als Näherung der heaviside
Funktion
    heavisideUsers(1000) over(20 seconds)
  ).protocols(httpConf)
)
```

4.1.2.4 Dynamische Daten

Ein weiteres Feature von Gatling ist das Einspeisen von dynamischen Daten, genannt Feeder. Mit den Feedern lassen sich zum Beispiel für jeden Nutzer unterschiedliche Eingabewerte für ein Suchfeld definieren. Gatling bietet mehre Möglichkeiten an in welchen Formaten die Daten angegeben werden können. Folgende Formate werden unterstützt [34]:

1. CSV
2. JSON
3. JDBC
4. Sitemap
5. Redis

Beispiel eines Skripts mit Feeder der Daten aus einer CSV Datei bekommt [35]:

```
CSV Datei:
searchCriterion,searchComputerName
Macbook,MacBook Pro
eee,ASUS Eee PC 1005PE

Code Snippet:

object Search {
// Erstellung eines Feeders aus einer CSV Datei
// .random Bestimmt das zufällig Daten ausgewählt werden
// Standardmäßig werden Daten in Reihe ausgelesen
  val feeder = csv("search.csv").random

  val search = exec(http("Home")
    .get("/"))
    .pause(1)
// Es wird ein Datensatz für den Nutzer aus der CSV geladen
// Dem Nutzer seiner Session werden die Attribute searchCriterion
// und searchComputerName zugwiesen
    .feed(feeder)
    .exec(http("Search")
// Ein Attribut wird als Parameter an die URL übergeben
    .get("/computers?f=${searchCriterion}")
// Mit Hilfe des CSS Selektors wird ein Teil der Antwort eingefangen
// ,auf dessen Inhalt überprüft und die Variable computerURL
// gespeichert
    .check(css("a:contains('${searchComputerName}')",
"href").saveAs("computerURL")))
    .pause(1)
    .exec(http("Select")
// Der zuvor gespeicherte Link wird aufgerufen
    .get("${computerURL}"))
    .pause(1)
}
```

Rohe Daten aus dem Feeder lassen sich auch konvertieren, beispielsweise kann ein String in ein Integer umgewandelt werden [36]:

```
csv("myFile.csv").convert {
  case ("attributeThatShouldBeAnInt", string) => string.toInt
}
```

Außerdem ist es möglich, dass alle Nutzer alle Daten verwenden sollen, hierfür ist der Feeder nicht geeignet. Dafür gibt es die Funktion `flattenMapIntoAttributes` [37]:

```
val records = csv("foo.csv").records

foreach(records, "record") {
  exec(flattenMapIntoAttributes("${record}"))
}
```

Des Weiteren lassen sich die eingefügten Daten auch basierend auf Informationen der Session filtern. Diese Funktion lässt sich mit dem Iterator der Feeder nicht anwenden, um dies umsetzen zu können muss man mit Hilfe von den Gatling-Parsern seine eigene Einfügelogik erstellen. Nachfolgend ein Beispiel mit zwei CSV Dateien, die mit Daten befüllt sind und anhand der Session Daten ein Wert aus den Dateien geholt wird [38]:

```
userProject.csv:
user, project
bob, aProject
sue, bProject

projectIssue.csv:
project,issue
aProject,1
aProject,12
bProject,64

feederexample.scala:
import io.gatling.core.feeder._
import scala.concurrent.forkjoin.ThreadLocalRandom

// index records by project
val recordsByProject: Map[String, IndexedSeq[Record[String]]] =
  csv("projectIssue.csv").records.groupBy{ record => record("project")
}

// convert the Map values to get only the issues instead of the full
records
val issuesByProject: Map[String, IndexedSeq[String]] =
  recordsByProject.mapValues{ records => records.map {record =>
record("issue")} }

// inject project
feed(csv("userProject.csv"))

  .exec { session =>
  // fetch project from  session
  session("project").validate[String].map { project =>

    // fetch project's issues
    val issues = issuesByProject(project)

    // randomly select an issue
    val selectedIssue =
issues(ThreadLocalRandom.current.nextInt(issues.length))
```

```
    // inject the issue in the session
    session.set("issue", selectedIssue)
  }
}
```

Hier wird ein „Issue“ zufällig aus der CSV Datei ausgewählt abhängig von dem „projekt“ das von der Session zurückgegeben wird.

4.1.2.5 Checks

Checks sind ein zusätzliches Feature, die sich gut mit Feedern verbinden lassen. Nachdem Daten aus dem Feeder geholt und in die Anwendung eingesetzt worden sind, kann mit Hilfe eines Checks überprüft werden, ob die Antwort vom Server den Erwartungen entspricht. Vorstellbar wäre auch, dass der Check die zu erwartenden Ergebnisse aus dem Feeder bekommt. Checks können zum Beispiel nach einem HTTP Request ausgeführt werden. Ein Beispiel für einen unkomplizierten Check nach einem HTTP Request [34]:

```
http("My Request").get("myUrl").check(status.is(200))
```

Des Weiteren kann der HTTP Header als auch der HTTP Body überprüft werden.

4.1.2.6 Assertions

Assertions können verwendet werden, um globale Statistiken, wie Reaktionszeiten und die Anzahl gescheiterter Requests zu überprüfen und zu überprüfen ob diese den Erwartungen entsprechen. Um Assertions für eine Simulation zu registrieren, muss folgendes getan werden:

```
setUp(scn).assertions(
  global.responseTime.max.lessThan(50),
  global.successfulRequests.percent.greaterThan(95)
)
```

Die Assertionsmethode kann beliebig viele Überprüfungen enthalten. Die Überprüfungen werden am Ende der Simulation ausgeführt, sollte diese scheitern gilt die Simulation als gescheitert. Für die Zusammensetzung einer Assertion existiert eine dedizierte Domain System Language. Sie besteht aus der Verkettung folgender Elemente [39]:

1. Scope
2. Statistic
3. Metric
4. Condition

Scope: Für den Scope können `global`, `forAll` oder `details(path)` verwendet werden

Statistic: Für die Statistic können `responseTime`, `allRequests`, `failedRequests`, `successfulRequests` oder `requestsPerSec` verwendet werden.

Matric: Für die Metric können für die Reponse Time `min`, `max`, `mean`, `stdDeav`, `percentile1`, `percentile2`, `percentile3` oder `percentile4` und für Anzahl der Requests `percent`, `permillion` oder `count` verwendet werden.

Condition: Für die Condition können `lessThan(threshold)`, `greaterThan(threshold)`, `between(thresholdMin, thresholdMax)`, `is(value)`, `in(squence)` verwendet werden. Mehrere Conditions lassen sich verketten.

Wenn Assertions definiert sind, werden zwei Ergebnisberichte im `js` Ordner erstellt. Diese liegen einmal im JSON und einmal im JUnit Format vor. Die JUnit Datei kann beispielweise für ein Jenkins Plugin verwendet werden. [39]

4.1.2.7 Schleifen

Um Wiederholungen im Scala Code zu vermeiden, gibt es die Möglichkeit Prozesse mit Schleifen zu realisieren. In der zu Beginn erstellten Simulation wurde mehrmals die Seite gewechselt. Die Anweisung zum Wechseln der Seiten lässt sich beispielsweise mit einer Schleife umsetzen [40]:

```
object Browse {

  val browse = repeat(5, "n") { // 1
    exec(http("Page ${n}")
      .get("/computers?p=${n}")) // 2
      .pause(1)
  }
}
```

Es ist aber auch vorstellbar mit Schleifen die Logik der Nutzerinjektion zu erweitern, um komplexe Tests zu erstellen.

4.1.2.8 Conditions

Sollen bestimmte Aktionen nur unter bestimmten Konditionen ausgeführt werden, kann dies mit den sogenannten Conditions umgesetzt werden. Eine Condition sieht wie folgt aus [41]:

```
doIf("${myBoolean}") {
  // wird ausgeführt wenn myBoolean wahr ist
  exec(http("...").get("..."))
}
```

4.1.3 Distributed Testing

Distributed Testing kann mit Hilfe der isolierten Prozesse umgesetzt werden, siehe Kapitel 4.1.2.2. Für jeden isolierten Prozess kann die Anzahl der virtuellen Nutzer und die Injektionsmethode eingestellt werden.

4.1.4 Ausführung der Tests

Zum Ausführen der Gatling Skripte muss das Shell-Skript ausgeführt werden, das sich im `bin` Verzeichnis von Gatling befinden. Wird das Skript ausgeführt sollte folgendes Menü erscheinen [42]:

```
Choose a simulation number:
   [0] computerdatabase.BasicSimulation
```

Hat man die gewünschte Simulation ausgewählt, wird diese ausgeführt. Während der Ausführung erhält man Feedback darüber welche Funktionen momentan ausgeführt werden, wie oft diese fehlgeschlagen sind oder erfolgreich waren und wie viele Requests warten, aktiv oder fertig sind.

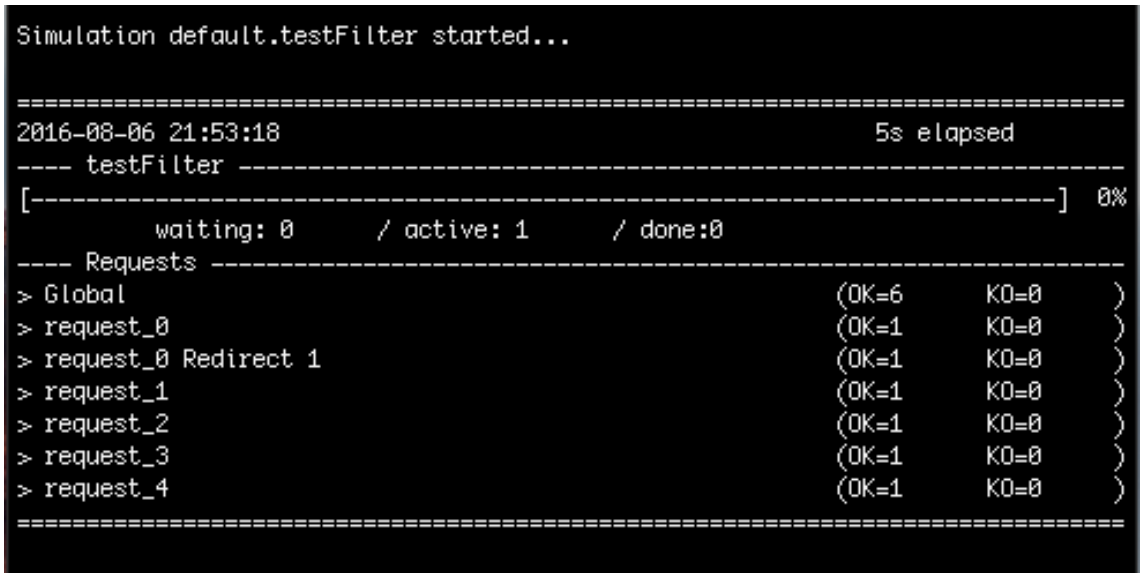

Abbildung 3 Beispielhafte Ausgabe von Gatling

Nach der Durchführung wird der Speicherort der Ergebnisse angezeigt, diese liegen mit grafischen Elementen im HTML Format vor, sowie Information darüber, wie schnell die Ergebnisse erstellt wurden. Die Ausgabe der grafischen Ergebnisse wird in Kapitel 4.1.7 beschrieben.

Abbildung 4 Informationsausgabe über den Speicherort

4.1.5 Konsolenparameter

Soll mehrmals der gleichen Test mit verschiedenen Nutzerzahlen ausgeführt werden, müsste für jede neue Nutzeranzahl das Skript angepasst werden. Gatling besitzt hierfür die Möglichkeit Parameter beim Ausführen der Skripte an das Skript weiterzugeben [43]:

```
JAVA_OPTS="-Dusers=500 -Dramp=3600"
```

Das Skript:

```
val nbUsers = Integer.getInteger("users", 1)
val myRamp  = java.lang.Long.getLong("ramp", 0L)
setUp(scn.inject(rampUsers(nbUsers) over (myRamp seconds)))
```

4.1.6 Remote Testing

Um größere Lasten herzustellen und ein Rechner dafür nicht ausreichend ist, muss man auf mehreren Rechnern Gatling ausführen. Gatling besitzt jedoch bisher keinen Cluster Modus der die Aufgabe automatisch übernimmt. Mit geringem Aufwand ist es aber trotzdem möglich sowas ähnliches wie einen Cluster Modus zu realisieren [44]:

1. Gatling auf mehreren Rechnern installieren mit den nötigen Simulationen und Daten.
2. Ferngesteuertes Ausführen der Skripte mit der Option `-nr` (keine Ergebnisse).
3. Einsammeln aller Simulations-Log-Dateien.
4. Umbenennen der einzelnen Logs, damit diese sich beim Verschieben nicht behindern.
5. Verschieben aller Logs in den `results` Ordner einer Gatling Installation.
6. Generieren der Ergebnisse erfolgt mit `-ro name-of-the-simulation-folder`. Gatling wählt alle verfügbaren log Dateien in dem Ordner aus, die dem Ausdruck `.*\.log` entsprechen.

Gatling stellt in ihrer Online Dokumentation ein Skript zur Verfügung, das diese Aufgaben erfüllen kann [44].

4.1.7 Auswertung der Ergebnisse

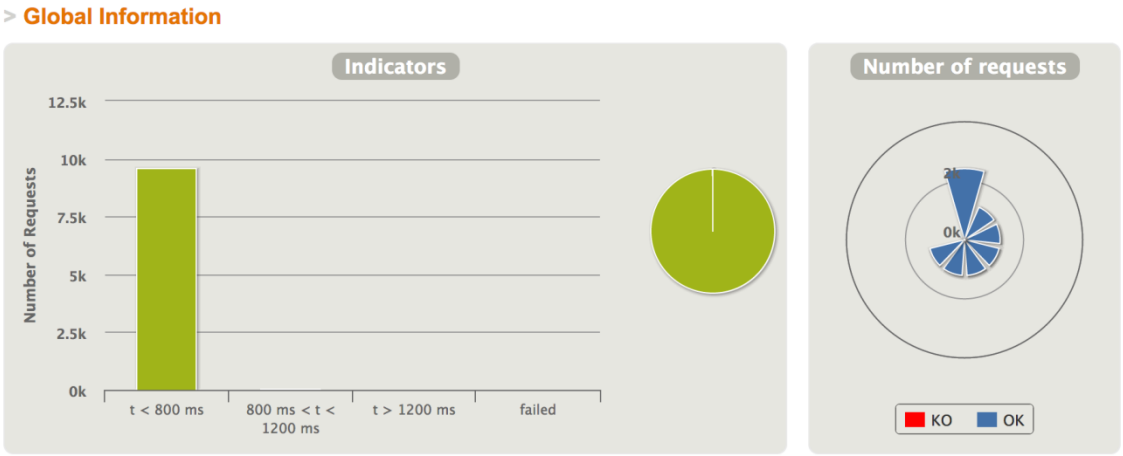

Abbildung 5 Gatling Reports Global information

Quelle: http://gatling.io/docs/2.2.2/_images/charts-indicators.png

Das „Number of Requests" Diagramm zeigt die Verteilung der Requests an, die innerhalb einem definierten Zeitraum erfüllt wurden. Der definierte Zeitraum ist dabei mög-

lichst sinnvoll einzustellen, möglichst nach den Kriterien, die die Anwendung erfüllen soll. Das rechte Diagramm zeigt die Anzahl der KO/OK Requests. Die angegebenen Reaktionszeiten für das linke Diagramm können in der `gatling.conf` angepasst werden.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions				Response Time (ms)						
	Total	OK	KO	% KO	Min	Max	Mean	Std Dev	95th pct	99th pct	Req/s
Global Information	9661	9659	2	0 %	94	1145	159	108	363	663	140.6
Home Redirect 1	2410	2410	0	0 %	94	1145	150	101	335	625	35.06
Search	1205	1205	0	0 %	97	965	156	106	366	656	17.53
Select	1205	1205	0	0 %	95	980	155	108	352	684	17.53
Page 0	1205	1205	0	0 %	100	1134	169	122	389	733	17.53
Page 1	1205	1205	0	0 %	100	1122	163	113	374	687	17.53
Page 2	1205	1205	0	0 %	100	992	165	110	367	677	17.53
Page 3	1205	1205	0	0 %	100	978	163	101	365	561	17.53
Form	7	7	0	0 %	98	512	194	148	453	500	0.10
Post Redirect 1	14	12	2	14 %	99	483	150	103	354	457	0.20

ERRORS

Error	Count	Percentage
status.is(201), but actually found 200	2	100.0 %

Abbildung 6 Gatling Statistics and Errors

Quelle: http://gatling.io/docs/2.2.2/_images/charts-statistics.png

Die Tabelle in Abbildung 4 zeigt Statistiken zu der kleinsten, größten und mittleren Abweichung der Reaktionszeiten an, wie viele Requests insgesamt stattfanden, die Anzahl wie viele erfolgreich waren, sowie einer Prozentangabe über die fehlgeschlagenen (KO) Requests. Die untere Tabelle zeigt einige Details zu den KO Requests an.

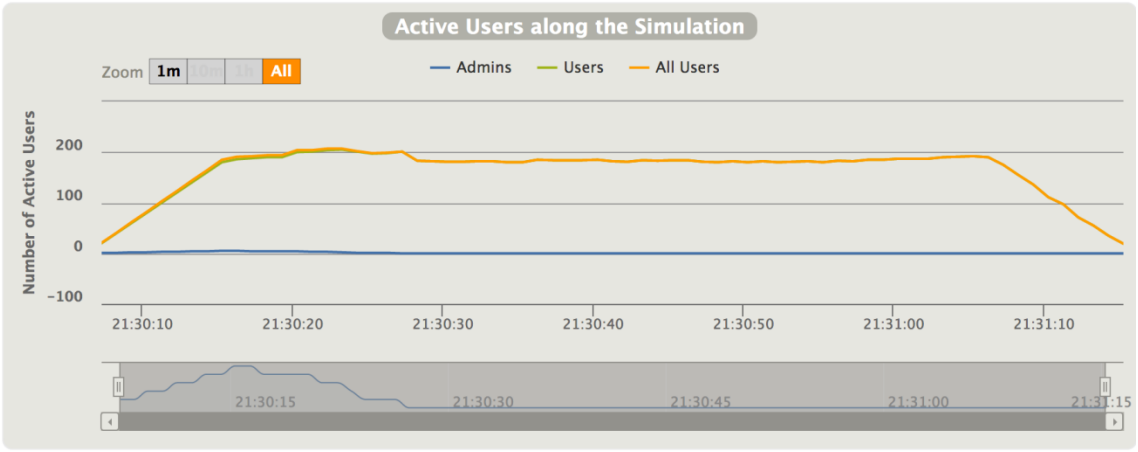

Abbildung 7: Aktive Nutzer über den Zeitraum

Quelle: http://gatling.io/docs/2.2.2/_images/charts-users.png

Das Diagramm aus Abbildung 5 zeigt die aktiven Nutzer die während der Simulation gleichzeitig auf der Anwendung aktiv waren.

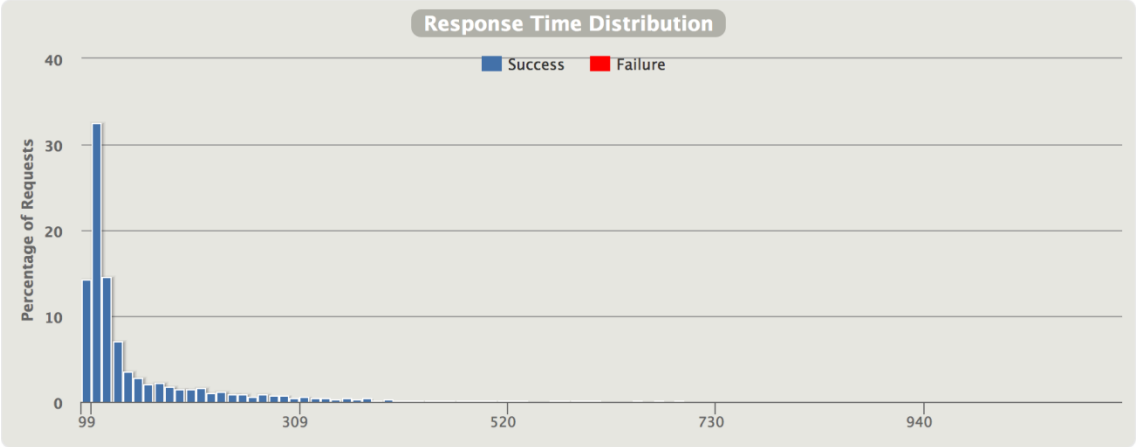

Abbildung 8: Response Time Verteilung

Quelle: http://gatling.io/docs/2.2.2/_images/charts-distrib.png

Das Diagramm aus Abbildung 6 zeigt die Verteilung der Reaktionszeiten an.

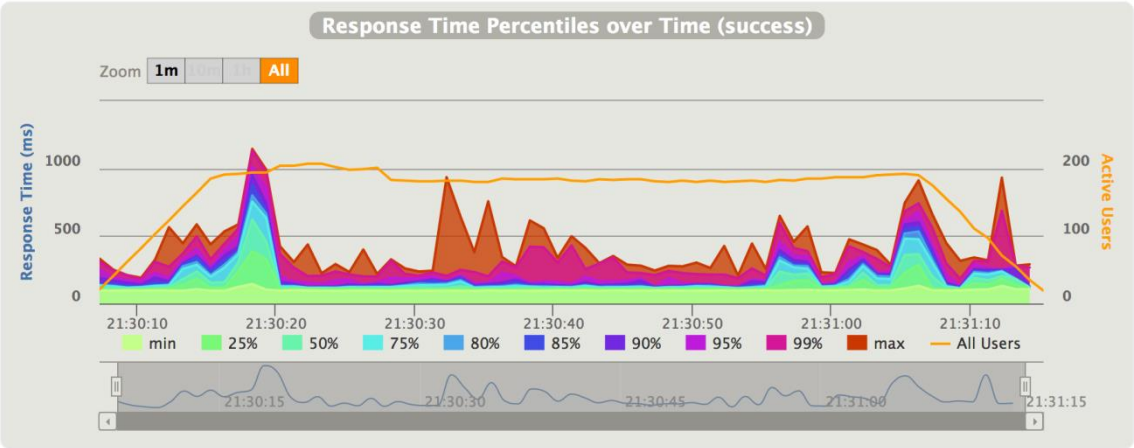

Abbildung 9: Response Time Perzentile über die Zeit

Quelle: http://gatling.io/docs/2.2.2/_images/charts-response-percentiles-per-sec.png

Das Diagramm aus Abbildung 7 gibt Auskunft darüber, wie viele Responses eine bestimmte Zeit benötigten in Prozent.

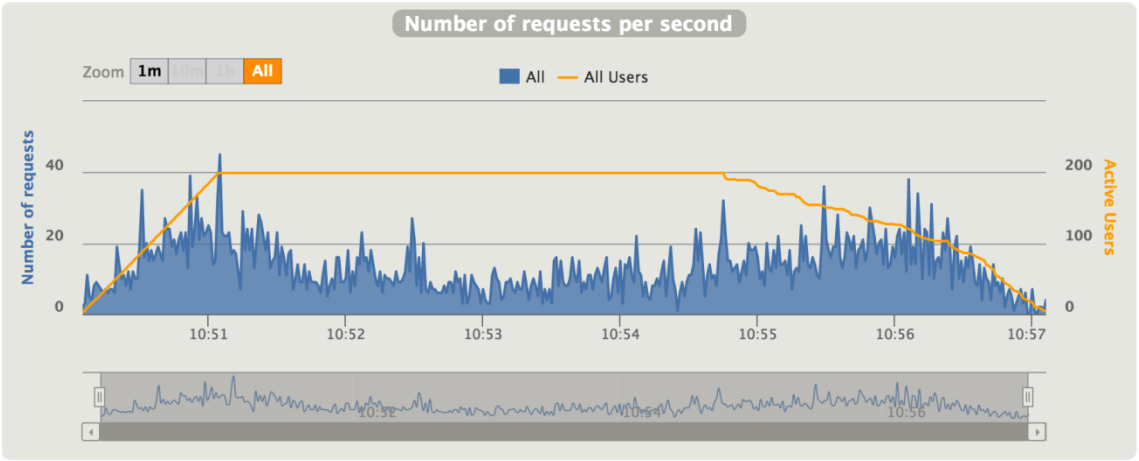

Abbildung 10: Anzahl der Requests über die Zeit

Quelle: http://gatling.io/docs/2.2.2/_images/charts-requests-per-sec.png

Das Diagramm aus Abbildung 8 zeigt die Anzahl der gesendeten Requests, die innerhalb einer Sekunde stattfanden an.

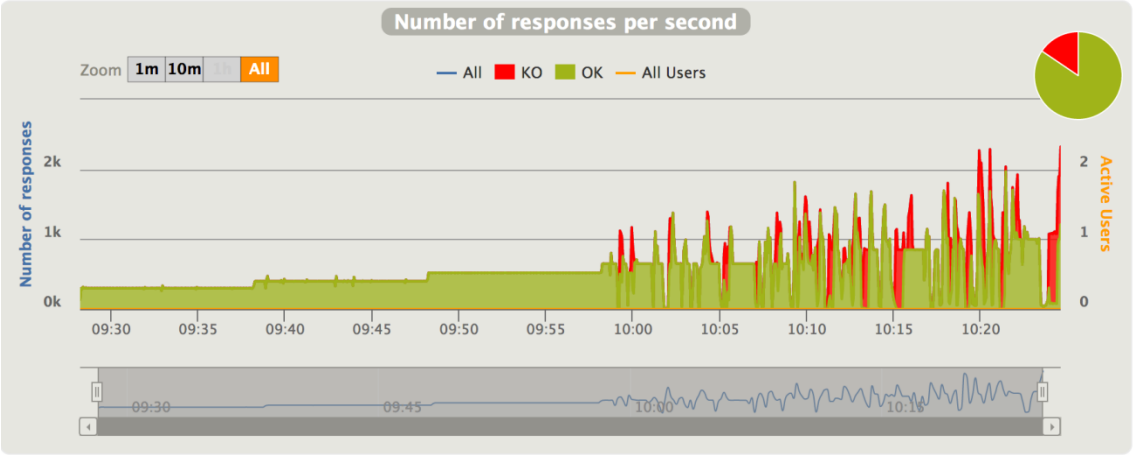

Abbildung 11: Responses pro Sekunde über die Zeit

http://gatling.io/docs/2.2.2/_images/charts-responses-per-sec.png

Das Diagramm aus Abbildung 9 zeigt die Anzahl der Responses, die innerhalb einer Sekunde vom Server geschickt wurden an.

Detail Diagramme:

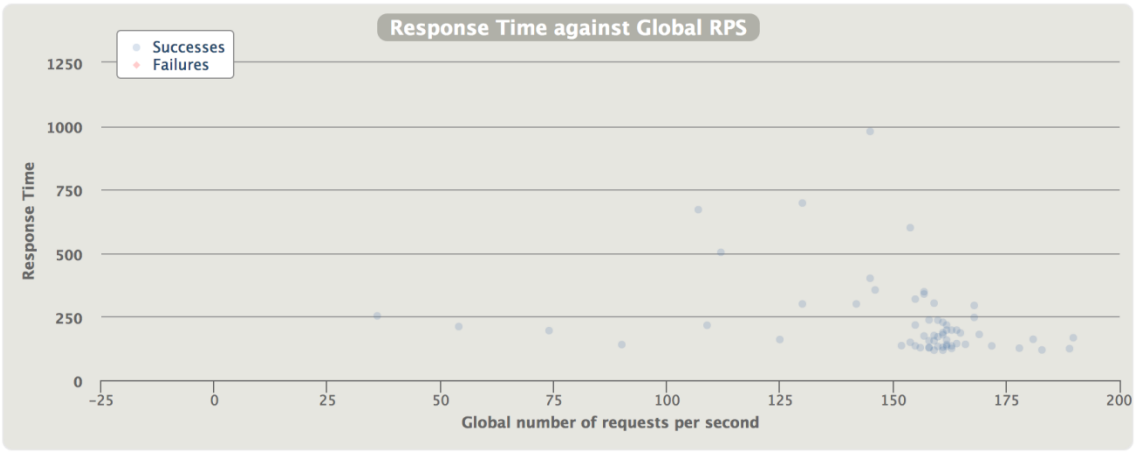

Abbildung 12: Response Zeiten im Vergleich zur Anzahl globaler Requests pro Sekunde

http://gatling.io/docs/2.2.2/_images/charts-response-time-global-rps.png

Das Diagramm aus Abbildung 10 zeigt die Verteilung der Reaktionszeiten aller Requests einer Request-Art an.

4.1.8 Realtime Monitoring

Mit Realtime Monitoring können Testergebnisse während der Ausführung betrachtet werden. Hierfür wird ein weiteres Tool benötigt, bekannte Vertreter sind Graphite und InfluxDB. Mittels dem Graphite Protokoll werden die Ergebnisse an das Graphite Backend geschickt. Damit die Messdaten versendet werden, muss die `Gatling.conf` folgendermaßen angepasst werden [45]:

```
data {
  writers = [console, file, graphite]
  reader = file

  graphite {
    host = "192.168.56.101"
    port = 2003
    #light = false              # only send the all* stats
    #protocol = "tcp"           # The protocol used to send data to
Carbon (currently supported : "tcp", "udp")
    #rootPathPrefix = "gatling" # The common prefix of all metrics
sent to Graphite
    #bufferSize = 8192          # GraphiteDataWriter's internal data
buffer size, in bytes
    #writeInterval = 1          # GraphiteDataWriter's write interval,
in seconds
  }
}
```

Es ist darauf zu achten, dass der korrekte Host und Port unter denen Graphite läuft eingestellt ist. Um serverseitige Systemlast aufzeichnen zu können wird ein weiteres Tool benötigt, das die Daten an das Graphite Backend schicken kann. [46]

4.1.9 HTTP/2 Unterstützung

Gatling unterstützt nach aktuellen Angaben keine HTTP/2 Verbindungen. [47] [48]

4.1.10 WebSocket Verbindungen

Es werden zwei Arten unterstützt, um Nachrichten mit dem WebSocket Protokoll zu verschicken. Die WebSocket Nachrichten können in Binär- oder in Textform versendet werden. [49] Ein Beispiel wie die WebSocket Verbindung realisiert werden kann in einem unkomplizierten Scala Testskript:

```
class WebSocket extends Simulation {

   val httpConf = http
      .wsBaseURL("ws://echo.websocket.org:80")
```

```
   val scn = scenario("WebSocket")
      .pause(1)
      .exec(ws("Connect WS").open("/"))
      .pause(1)
      .exec(ws("Say Hello WS")
          .sendText("Hello")
          .check(wsAwait.within(30).until(1).regex("Hello"))
        )
      .pause(1)
      .exec(ws("Close WS").close)

   setUp(scn.inject(atOnceUsers(10)).protocols(httpConf)
}
```

4.2 JMeter

JMeter lässt sich komplett über die GUI steuern. Das besondere an JMeter ist dessen modularer Aufbau. Das bedeutet, dass alle eingebauten Funktionen als Plugin integriert sind und von JMeter als auch unabhängigen freiwilligen Mitwirkenden entwickelt werden können.

Schlüssel Funktionen von JMeter [29]:

1. Lauffähig auf allen Plattformen die Java unterstützen.
2. Skalierbarkeit: Ist mehr Leistung nötig kann JMeter im Verteilungsmodus verwendet werden.
3. Unterstützt zahlreiche Protokolle: HTTP(S), SMTP, POP3, IMAP, LDAP, JDBC, FTP, JMS, SOAP/REST, TCP, native Kommandos oder Shell Skripte.
4. Durch die Implementierung von Pre- und Post-Processors sind komplexe Tests möglich.
5. Mehrere Assertion Module um Kriterien festzulegen.
6. Mehrere Listener um die Ergebnisse zu visualisieren.
7. Multithreading möglich mit Thread Groups.
8. Seit Version 3.0 die Möglichkeit Test Ergebnisse als HTML auszugeben mit „Dasboard Report“.
9. Realtime Monitoring fähig.

4.2.1 Erstellung der Szenarien mit der GUI

In JMeter werden die Szenarien frei von Code in der GUI erstellt, über verschiedene Elemente die hinzugefügt und konfiguriert werden. JMeter verfügt auch wie Gatling einen HTTP Recorder, der die HTTP Requests aufzeichnet und in den Testplan integriert, aber ohne Aufzeichnung der Nutzer Denkzeiten.

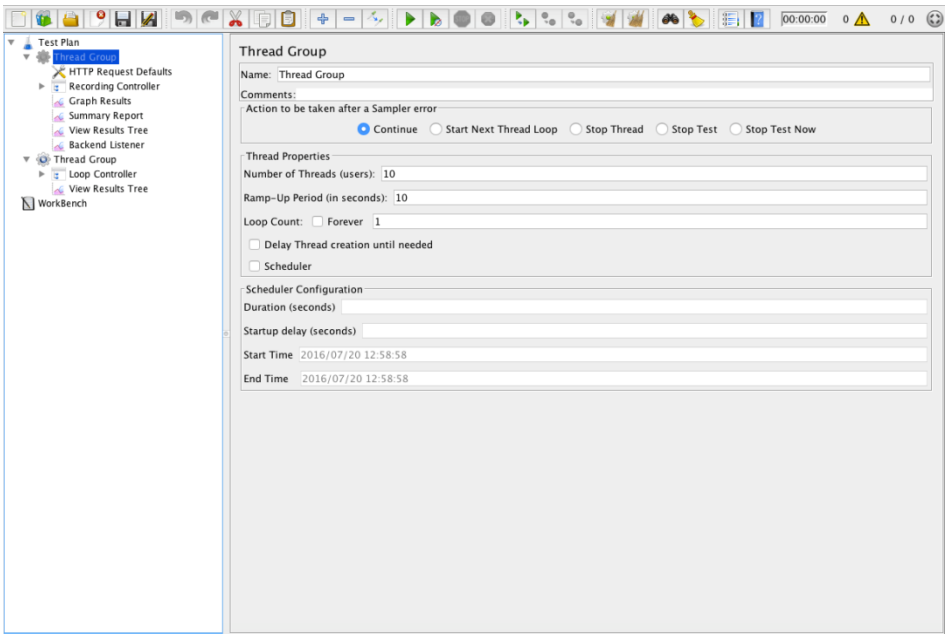

Abbildung 13: JMeter GUI

4.2.1.1 HTTP(S) Script Recorder

JMeter besitzt einen Recorder mit dessen Hilfe die Interaktion mit dem Browser aufgezeichnet werden kann. Dieser verwendet einen Proxy über den die Daten gesendet werden, um die Aktionen aufzeichnen zu können. Der Recorder zeichnet keine Reaktionszeiten auf, er zeichnet lediglich die HTTP Requests auf, die ausgeführt wurden.

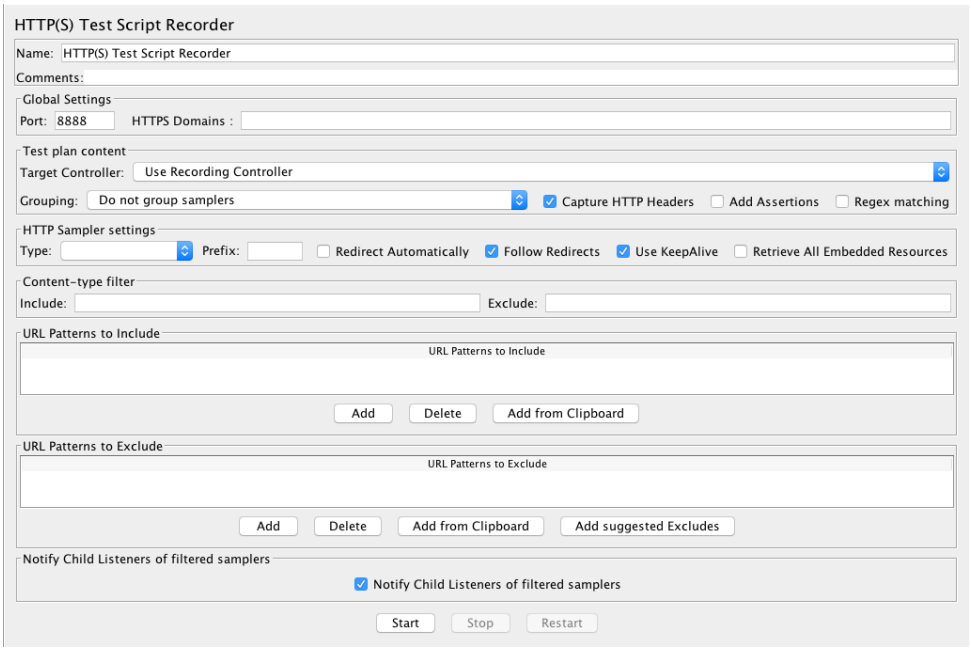

Abbildung 14: Konfiguration des Recorders

Hat man erfolgreich die nötigen HTTP Requests aufgezeichnet sieht das wie folgt aus:

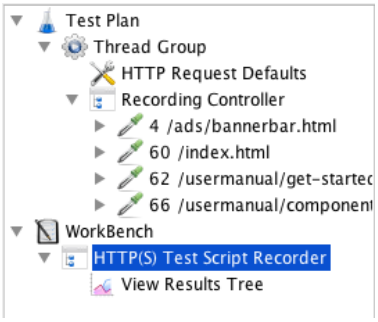

Abbildung 15: Aufgezeichnete HTTP Requests

Nach dem Aufzeichnen der Requests können nun bereits erklärte Elemente hinzugefügt werden, um einen komplexeren Test zu erstellen.

4.2.1.2 Thread Groups

Im Thread Groups Element lässt sich die Anzahl der Wiederholungen einstellen, die ein Testplan ausführen soll. Es lassen sich die Anzahl der Threads, die Anlaufzeit (in Sekunden) und die Anzahl der Wiederholungen einstellen. Zusätzlich kann eingestellt werden, wann der Test automatisch ausgeführt werden soll, die Dauer und mit welcher Verzögerung dieser ausgeführt werden soll. Außerdem kann das Verhalten eingestellt werden, was geschehen soll falls ein Fehler während der Ausführung auftritt.

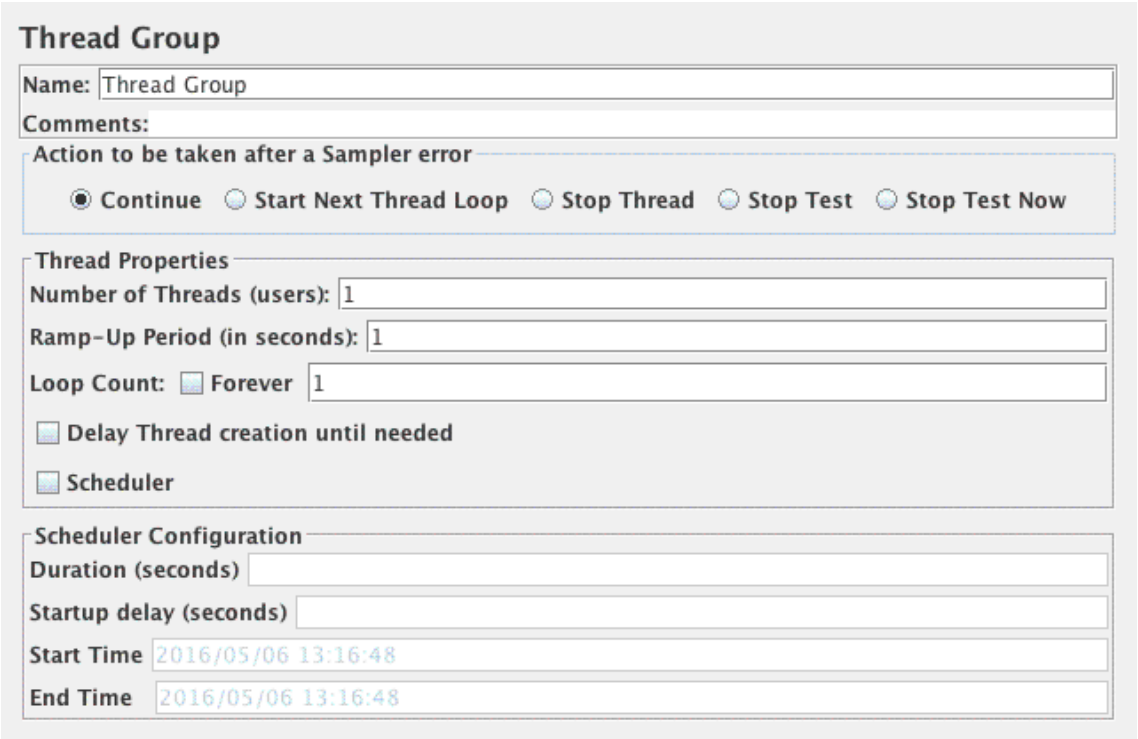

Abbildung 16: Thread Group Konfiguration

Quelle: http://jmeter.apache.org/usermanual/build-web-test-plan.html

4.2.1.3 Sampler

Die Sampler sind für die eigentliche Arbeit zuständig. Sie führen die Aktionen auf die Anwendung an und generieren dabei Ergebnisse, die von den Listener ausgewertet und angezeigt werden können. JMeter bietet folgende Sampler an [50]:

- FTP Request
- HTTP Request
- JDBC Request
- Java Request
- SOAP/XML-RPC Request
- LDAP Request
- LDAP Extended Request
- Access Log Sampler
- BeanShell Sampler
- BSF Sampler
- JSR223 Sampler
- TCP Sampler
- JMS Publisher
- JMS Subscriber
- JMS Point-to-Point
- JUnit Request
- Mail Reader Sampler
- Test Action
- SMTP Sampler
- OS Process Sampler
- MongoDB Script (Veraltet)

4.2.1.4 Pre-/Post-Processors

Pre-Processors bieten weitere Möglichkeiten an den Sampler anzupassen. Beispielweise lassen sich Nutzer Parameter angeben. Folgende Pre-Processors stehen zur Verfügung [51]:

- HTML Link Parser
- HTTP URL Re-writing Modifier

- User Parameters
- BeanShell PreProcessor
- BSF PreProcessor
- JSR223 PreProcessor
- JDBC PreProcessor
- RegEx User Parameters
- Sample Timeout

Post-Processors werden nach der Verwendung des Samples ausgeführt, jedoch vor den Assertions. Sie extrahieren Informationen aus der Response, diese werden in eine JMeter Variable geschrieben, welche das Assertion-Element auslesen und überprüfen kann. Möchte man auf die Assertion-Ergebnisse Zugriff haben, sollte man einen Listener verwenden. Folgende Post-Processors stehen zur Verfügung [52]:

- Regular Expression Extractor
- CSS/JQuery Extractor
- XPath Extractor
- Result Status Action Handler
- BeanShell PostProcessor
- BSF PostProcessor
- JSR223 PostProcessor
- JDBC PostProcessor
- JSON Path PostProcessor

4.2.1.5 Logic Controllers

Die Controller geben an, in welcher Reihenfolge die Sampler ausgeführt werden sollen. Folgende Controller stehen zur Verfügung [53]:

- Simple Controller
- Loop Controller
- Once Only Controller
- Interleave Controller
- Random Controller
- Random Order Controller
- Throughput Controller

- Runtime Controller
- If Controller
- While Controller
- Switch Controller
- ForEach Controller
- Module Controller
- Include Controller
- Transaction Controller
- Recording Controller
- Critical Section Controller

4.2.1.6 Listeners

Listener geben die Testergebnisse aus. Jeder Listener wird immer am Ende eines Bereiches in dem er eingefügt wurde ausgeführt. Sie zeigen die Ergebnisse der Sampler an. Folgende Listener stehen zur Verfügung [54]:

- Sample Result Save Configuration
- Graph Results
- Spline Visualizer (veraltet)
- Assertion Results
- View Results Tree
- Aggregate Report
- View Results in Table
- Simple Data Writer
- Monitor Results
- Distribution Graph (veraltet)
- Aggregate Graph
- Response Time Graph
- Mailer Visualizer
- BeanShell Listener
- Summary Report
- Save Responses to a file
- BSF Listener

- JSR223 Listener
- Generate Summary Results
- Comparison Assertion Visualizer
- Backend Listener

4.2.1.7 Configuration Elements

Hier können Daten festgelegt werden, die von Samplern verwendet werden sollen, wie zum Beispiel Daten aus einer CSV Datei oder einer Datenbank laden, oder Angaben zu Login Daten machen. Konfigurationen werden immer zu Beginn eines Bereiches, in denen sie sich befinden, ausgeführt. Folgende Configuration Elements stehen zur Verfügung [55]:

- CSV Data Set Config
- FTP Request Defaults
- DNS Cache Manager
- HTTP Authorization Manager
- HTTP Cache Manager
- HTTP Cookie Manager
- HTTP Request Defaults
- HTTP Header Manager
- Java Request Defaults
- JDBC Connection Configuration
- Keystore Configuration
- Login Config Element
- LDAP Request Defaults
- LDAP Extended Request Defaults
- TCP Sampler Config
- User Defined Variables
- Random Variable
- Counter
- Simple Config Element
- MongoDB Source Config (DEPRECATED)

4.2.1.8 Assertions

Assertions werden verwendet, um die Antworten des Servers zu überprüfen. Stimmen die Antworten nicht mit dem festgelegten Erwartungswert überein gilt der Test als fehlgeschlagen. Sie werden nach jedem Sampler, im Bereich in dem sie sich befinden, ausgeführt. Ein Beispiel eines möglichen Assertion Element ist die Reponse Assertion:

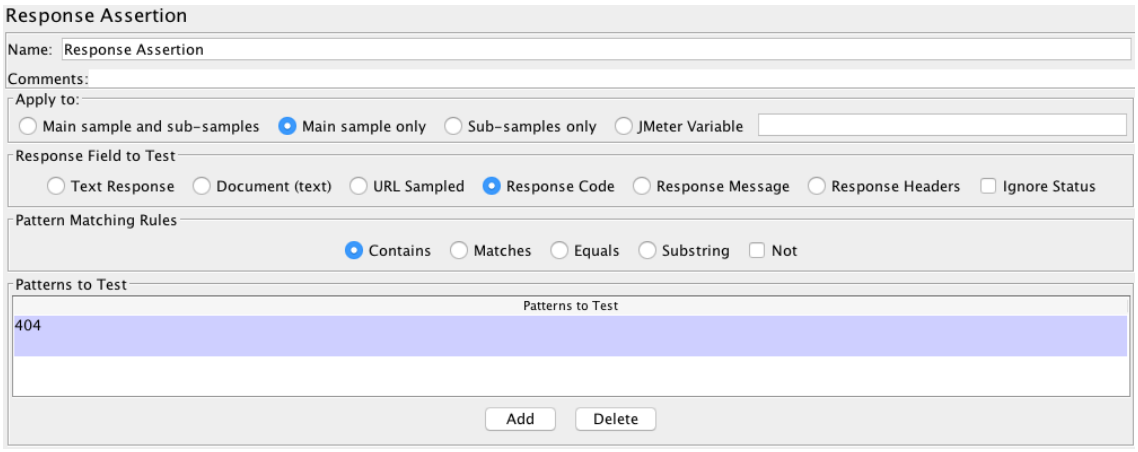

Abbildung 17: Response Assertion

In Abbildung 12 wird der Response Code nach dem Text „404“ untersucht. Sollte dieser Text nicht im Response Code enthalten sein, sowie angegeben, gilt der Test als fehlgeschlagen. Das Ergebnis sieht wie folgt aus:

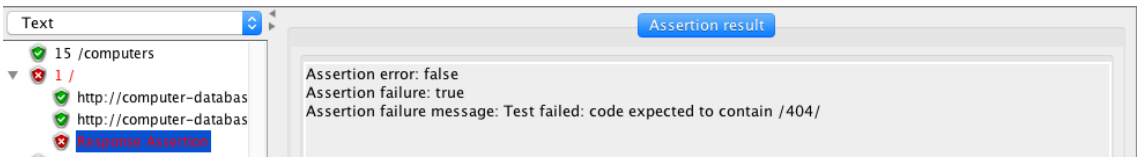

Abbildung 18: Das Ergebnis einer Assertion

Folgende Assertion-Elemente stehen zur Verfügung [56]:

- Response Assertion
- Duration Assertion
- Size Assertion
- XML Assertion
- BeanShell Assertion
- MD5Hex Assertion
- HTML Assertion
- XPath Assertion
- XML Schema Assertion

- BSF Assertion
- JSR223 Assertion
- Compare Assertion
- SMIME Assertion

4.2.1.9 Timer

Timer werden verwendet um die Threads zu pausieren, damit lassen sich beispielsweise Denkzeiten der Nutzer simulieren. Sie werden vor jedem Sampler ausgeführt in dessen Bereich sie sich befinden. Sollten mehrere Timer sich in einem Bereich befinden, werden alle Timer vor jedem einzelnen Sampler in dem Bereich ausgeführt. Um einen Timer einen Sampler zuzuweisen sollte der Timer als dessen Kind-Element hinzugefügt werden.

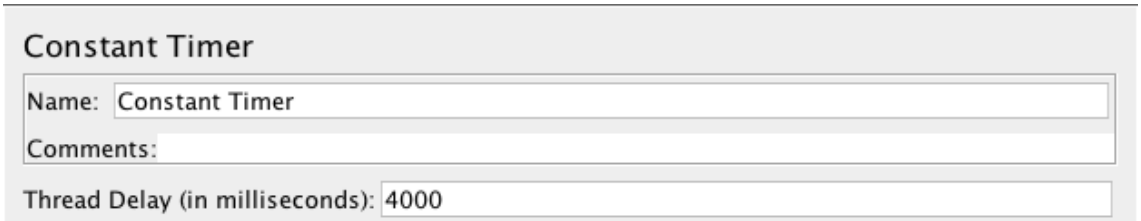

Abbildung 19: Constant Timer Konfiguration

Folgende Timer stehen zur Verfügung [57]:

- Constant Timer
- Gaussian Random Timer
- Uniform Random Timer
- Constant Throughput Timer
- Synchronizing Timer
- BeanShell Timer
- BSF Timer
- JSR223 Timer
- Poisson Random Timer

4.2.2 Distributed Testing

Distributed Testing kann beispielweise über das Verwenden des Thread Groups Elements umgesetzt werden, indem mehrere Thread Groups im Testplan und mit unterschiedlichen Einstellungen erstellt werden.

4.2.3 Ausführung der Tests

Durch auswählen des Start-Knopfs in der oberen Leiste der JMeter GUI können die Tests gestartet werden.

4.2.4 Konsolenparameter mit dem Non-GUI Modus

Ohne GUI lassen sich JMeter Testpläne ausführen aber nicht erstellen. Der Non-GUI Modus ist relevant, wenn man eine Maschine zum testen verwendet, die selber keine GUI besitzt, wie zum Beispiel ein Server der Last auf eine Anwendung generieren sollen, oder man Ressourcen einsparen möchte durch nicht Verwendung der GUI. Folgende Befehle existieren für den Non-GUI Modus [58] [59]:

```
Example console command:
jmeter -n -t D:\TestScripts\script.jmx -l
D:\TestScripts\scriptresults.jtl

-n
    This specifies JMeter is to run in non-gui mode
-t
    [name of JMX file that contains the Test Plan].
-l
    [name of JTL file to log sample results to].
-j
    [name of JMeter run log file].
-r
    Run the test in the servers specified by the JMeter property "remote_hosts"
-R
    [list of remote servers] Run the test in the specified remote servers
-g
    [path to CSV file] generate report dashboard only
-e
    generate report dashboard after load test
-o
    output folder where to generate the report dashboard after load test.
    Folder must not exist or be empty
    The script also lets you specify the optional firewall/proxy server information:
-H
    [proxy server hostname or ip address]
-P
    [proxy server port]
```

4.2.5 Remote Testing

Sollte die Leistung einer einzelnen Testmaschine nicht ausreichen, ist es mit JMeter möglich mehrere Maschinen als Testserver und eine Maschine als Verwalter dieser zu verwenden. Hierfür muss auf jeder Maschine die gleiche JMeter Version installiert werden. Es wird auch empfohlen die selbe Java Version auf allen Injektor Maschinen laufen

zu lassen. Außerdem müssen sich alle Maschinen im selben Subnetz befinden. Auf den Server Maschinen muss jedoch `Jmeter-server.bat` ausgeführt werden und nicht die normale GUI mit `Jmeter.bat`. [60] Des Verwalters `jmeter.properties` muss um die IP Adressen der Testserver unter der Eigenschaft `remote_hosts` erweitert werden. Als nächstes kann die Konfiguration mit der GUI überprüft werden, diese sollte nur für das Debugging verwendet werden und nicht für die spätere Testausführung, damit die Performance von JMeter sich nicht verschlechtert. Hat man die Konfiguration überprüft, muss JMeter im Non-GUI Modus mit Angabe des `-r` Parameters oder Aufführen der einzelnen Serveradressen, ausgeführt werden. Ein beispielhafter Konsolenbefehl kann wie folt aussehen: [61]

```
jmeter -Rhost1,127.0.0.1,host2
```

4.2.6 Auswertung der Ergebnisse

Für die Auswertung der Ergebnisse werden für gewöhnlich die Listener verwendet. Seit Version 3.0 ist es möglich die Testergebnisse in Graphen im HTML Format generieren zu lassen. Um diese Funktion freizuschalten muss man die `user.properties` im `bin` Verzeichnis um folgende Zeilen erweitert [62] [63]:

```
# Configure this property to change the report title
#jmeter.reportgenerator.report_title=Apache JMeter Dashboard

# Change this parameter if you want to change the granularity of over
time graphs.
#jmeter.reportgenerator.overall_granularity=60000

 Change this parameter if you want to change the granularity of
Response time distribution
# Set to 500 ms by default
#jmeter.reportgenerator.graph.responseTimeDistribution.property.set_gr
anularity=500

# Change this parameter if you want to override the APDEX satisfaction
threshold.
jmeter.reportgenerator.apdex_satisfied_threshold=1500

# Change this parameter if you want to override the APDEX tolerance
threshold.
jmeter.reportgenerator.apdex_tolerated_threshold=3000

# Sets the destination directory for generated html pages, it is
better to change it for every generation
# This will override the value set through -o command line option
# jmeter.reportgenerator.exporter.html.property.output_dir=/tmp/test-
report
```

```
# Indicates which graph series are filtered (regular expression)
# In the below example we filter on Search and Order samples
# Note that the end of the pattern should always include (-success|-
failure)?
# Transactions per second suffixes Transactions with "-success" or "-
failure" depending
# on the result
#jmeter.reportgenerator.exporter.html.series_filter=((^Search)|(^Order
))(-success|-failure)?

# Indicates whether series filter apply only on sample series
jmeter.reportgenerator.exporter.html.filters_only_sample_series=true
```

Nachdem man die richtige Konfiguration eingestellt hat, muss man per Konsole in das `bin` Verzeichnis navigieren und folgenden Befehl ausführen [64]:

```
jmeter -g /Pfad/zur/ergebnis/Datei -o
/Dashboard/Verzeichnis/Speicherpunkt
```

Danach sollte im Zielverzeichnis das generierte Dashboard vorhanden sein. Führt man die beinhaltende `index.html` aus wird im Browser folgendes angezeigt:

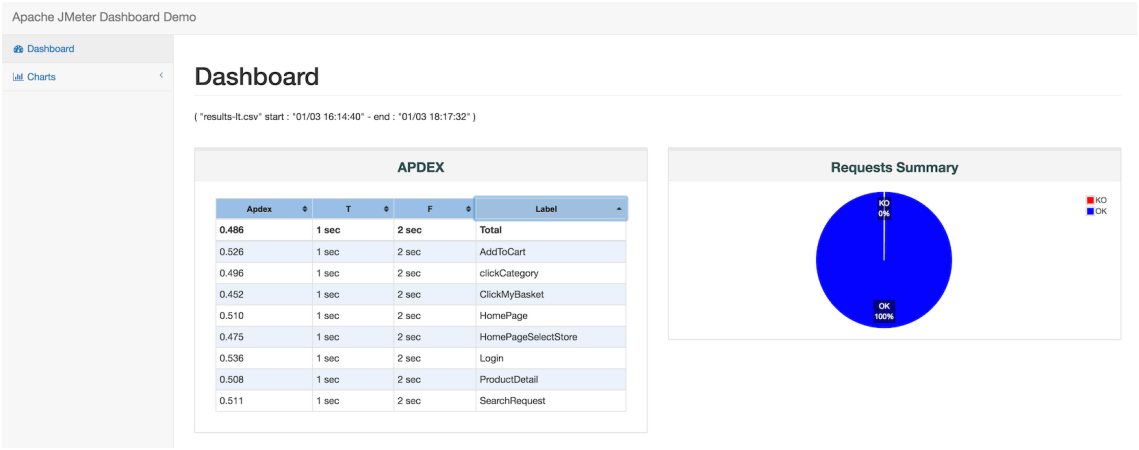

Abbildung 20: Startseite des Dashboard Reports

Im Dashboard lassen sich folgende Statistiken anzeigen [65]:

1. Erfolgreiche und fehlgeschlagene Transaktionen in Prozent.
2. Eine Tabelle, die alle Messdaten der Transkationen beinhaltet mit drei konfigurierbaren Perzentil-Spalten.
3. Zusammenfassung aller aufgetretenen Fehler, im Verhältnis zu allen Requests in Prozent.

4. Vergrößerbare Diagramme mit der Möglichkeit einzelne Transaktionen anzeigen und ausblenden zu lassen:
    - Reaktionszeiten über die Zeit
    - Datendurchsatz über die Zeit in Bytes
    - Latenzen über Zeit
    - Aufrufe pro Sekunde
    - Response Codes pro Sekunde
    - Transaktionen pro Sekunde
    - Reaktionszeit vs. Requests pro Sekunde
    - Latenz vs. Requests pro Sekunde
    - Reaktionszeiten in Prozent
    - Aktive Threads über die Zeit
    - Zeiten vs. Threads
    - Reaktionszeitverteilung

4.2.7 Realtime Monitoring

JMeter besitzt, wie Gatling, die Fähigkeit Echtzeit Ergebnisse an ein Backend zu senden. Auch JMeter gibt Graphite als Tool für Echtzeit Überwachung an. Damit die Messdaten an das Backend gesendet werden können, muss das BackendListener Element hinzugefügt und konfiguriert werden.

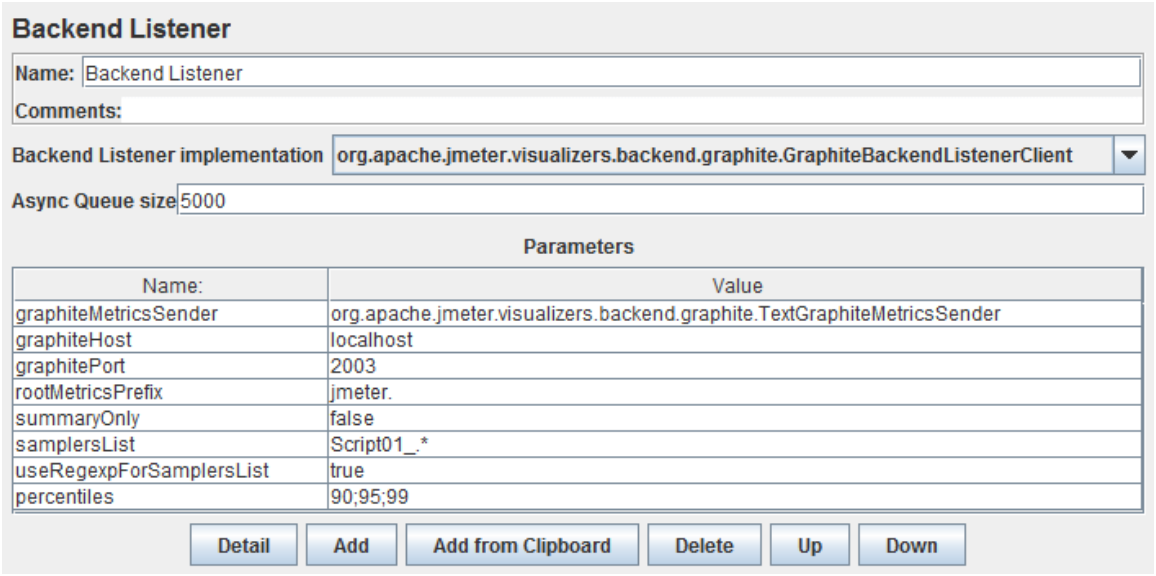

Abbildung 21: Backend Listener Element

Quelle: http://jmeter.apache.org/images/screenshots/backend_listener.png

4.2.8 HTTP/2 Unterstützung

JMeter besitzt momentan keine HTTP/2 Unterstützung. [66]

4.2.9 WebSocket Verbindungen mit dem Blazemeter Plugin

JMeter unterstützt in seiner Basisform nicht das 2011 standardisierte WebSocket Protokoll. Durch den modularen Aufbau von JMeter ist es möglich das Blazemeter Plugin der Installation hinzuzufügen mit dem sich WebSocket Verbindungen testen lassen, dass sich wie andere JMeter Elemente standartmäßig konfigurieren lässt. [67] In der Konfiguration ist es möglich Parameterwerte zu übergeben und Request Daten mitzuschicken.

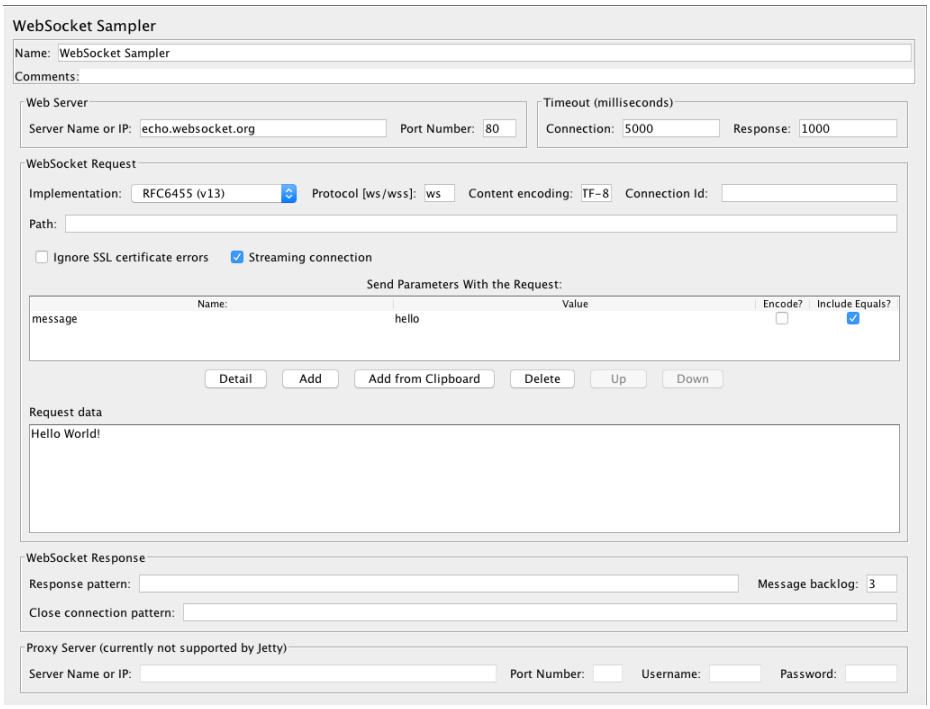

Abbildung 22: WebSocket Sampler

4.3 The Grinder

Nachfolgend in dieser Arbeit Grinder genannt, lässt sich teilweise über eine grafische Oberfläche bedienen. Das Starten von Grinder geschieht über die Ausführung von Shell Skripte, die vorher erstellt werden müssen. Diese können über die Konsole ausgeführt werden. Grinder ist eine Java Anwendung, wodurch gegeben ist, dass es auf vielen Plattformen verwendet werden kann. Damit die Skripte ausgeführt werden können, muss zuerst der Agent gestartet werden. Anschließend wird der Test über die GUI gestartet. Durch die Aufteilung in GUI und Agenten lassen sich mehrere Load Injector Maschinen steuern, indem auf verschiedenen Maschinen die Agenten gestartet werden. Ein Agent kann aus mehreren Workern bestehen und deren Verhalten steuern. Folgende Funktionen bietet Grinder an [29] [68]:

1. TCP-Proxy zur Aufzeichnung der Netz Aktivität und Integrierung in das Testskript.
2. Unterstützung für verteiltes Testen, das mit steigender Agentenzahl skaliert.
3. Jython oder Closure als Skriptsprache. Jython basiert auf der Java Implementierung von Python
4. Zugriff auf die Java API.
5. Flexibel durch Parametrisierung. Es können Testdaten während einem Testlauf erstellt werden sowie externe Daten aus einer Datenbank oder Textdatei eingebunden werden.
6. Post-processing und Assertions, welche Zugriff auf Ergebnisse haben.
7. Unterstützung für mehrere Protokolle (HTTP, TCP, SSL).
8. Unterstützung folgender Lasttests [69]:
    - Load Testing
    - Capacity Testing
    - Functional Testing
    - Stress Testing

4.3.1 Voraussetzungen

Bevor Grinder überhaupt gestartet werden kann, müssen folgende Shell-Skripte erstellt werden [70]:

1. setGrinderEnv.sh/cmd
    - Setzt die Umgebungsvariablen für die Grinder Installation

```
Unix:
#!/usr/bin/ksh
GRINDERPATH=<em>(full path to grinder installation directory)</em>
GRINDERPROPERTIES=<em>(full path to
grinder.properties)</em>/grinder.properties
CLASSPATH=$GRINDERPATH/lib/grinder.jar:$CLASSPATH
JAVA HOME=<em>(full path to java installation directory)</em>
PATH=$JAVA_HOME/bin:$PATH
export CLASSPATH PATH GRINDERPROPERTIESexport CLASSPATH PATH
GRINDERPROPERTIES

Windows:
set GRINDERPATH=<em>(full path to grinder installation directory)</em>
set GRINDERPROPERTIES=<em>(full path to
grinder.properties)</em>\grinder.properties
set CLASSPATH=%GRINDERPATH%\lib\grinder.jar;%CLASSPATH%
set JAVA_HOME=<em>(full path to java installation directory)</em>
PATH=%JAVA_HOME%\bin;%PATH%
```

2. startAgent.sh/cmd
    - Startet den Agenten

```
Unix:
#!/usr/bin/ksh
. <em>(path to setGrinderEnv.sh)</em>/setGrinderEnv.sh
java -classpath $CLASSPATH net.grinder.Grinder $GRINDERPROPERTIES

Windows:
call <em>(path to setGrinderEnv.cmd)</em>\setGrinderEnv.cmd
echo %CLASSPATH%
java -classpath %CLASSPATH% net.grinder.Grinder %GRINDERPROPERTIES%
```

3. startConsole.sh/cmd
    - Öffnet die grafische Oberfläche von Grinder

```
Unix:
#!/usr/bin/ksh
. <em>(path to setGrinderEnv.sh)</em>/setGrinderEnv.sh
java -classpath $CLASSPATH net.grinder.Console

Windows:
call <em>(path to setGrinderEnv.cmd)</em>\setGrinderEnv.cmd
java -classpath %CLASSPATH% net.grinder.Console
```

4. startProxy.sh/cmd
    - Startet den TCPProxy Recorder

```
Unix:
#!/usr/bin/ksh
. <em>(path to setGrinderEnv.sh)</em>/setGrinderEnv.sh
java -classpath $CLASSPATH net.grinder.TCPProxy -console -http >
grinder.py

Windows:
call <em>(path to setGrinderEnv.cmd)</em>\setGrinderEnv.cmd
java -classpath %CLASSPATH% net.grinder.TCPProxy -console -http >
grinder.py
```

4.3.2 Die GUI

Die Gestaltung der GUI von Grinder ist simpel gestaltet, rechts befinden sich vier Tabs. Diese Tabs sind: Graphen, Ergebnisse, Prozesse und Skripte, unter denen hin und her geschaltet werden kann. Links befinden sich einige Einstellungen über die Aufnahme der Ergebnisse.

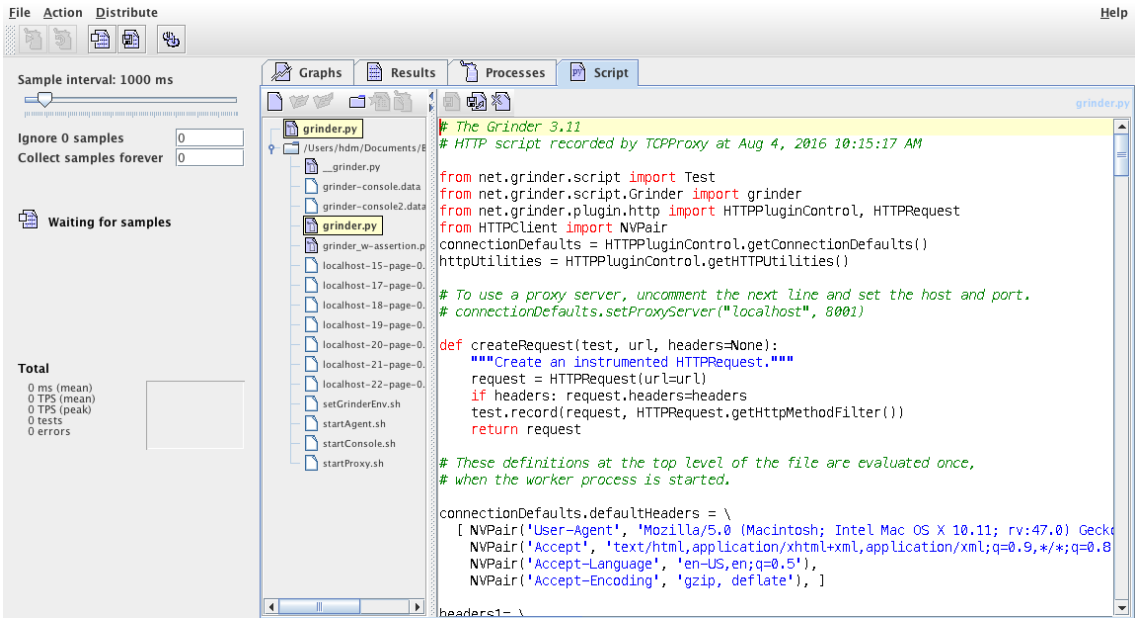

Abbildung 23: The Grinder GUI

4.3.3 Erstellung der Use-Case Skripte

Für die Erstellung der Use-Case Skripte wird die Programmiersprache Jython verwendet, die auf der Java Implementierung von Python basiert. Seit Version 3.6 wird außerdem die Sprache Clojure unterstützt, auf die in dieser Arbeit nicht weiter eingegangen wird. [71] Ein Beispiel für ein Jython Skript mit einem einfachen Request auf die Startseite von `http://computer-database.gatling.io/`:

```
# The Grinder 3.11
# HTTP script recorded by TCPProxy at Aug 1, 2016 4:14:09 PM
from net.grinder.script import Test
from net.grinder.script.Grinder import grinder
from net.grinder.plugin.http import HTTPPluginControl, HTTPRequest
from HTTPClient import NVPair
connectionDefaults = HTTPPluginControl.getConnectionDefaults()
httpUtilities = HTTPPluginControl.getHTTPUtilities()

# To use a proxy server, uncomment the next line and set the host and
port.
# connectionDefaults.setProxyServer("localhost", 8001)

def createRequest(test, url, headers=None):
    """Create an instrumented HTTPRequest."""
    request = HTTPRequest(url=url)
    if headers: request.headers=headers
    test.record(request, HTTPRequest.getHttpMethodFilter())
    return request

# These definitions at the top level of the file are evaluated once,
# when the worker process is started.

connectionDefaults.defaultHeaders = \
  [ NVPair('User-Agent', 'Mozilla/5.0 (Macintosh; Intel Mac OS X
10.11; rv:47.0) Gecko/20100101 Firefox/47.0'),
    NVPair('Accept',
'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'),
    NVPair('Accept-Language', 'en-US,en;q=0.5'),
    NVPair('Accept-Encoding', 'gzip, deflate'), ]

url0 = 'http://computer-database.gatling.io:80'

request101 = createRequest(Test(101, 'GET /'), url0)

class TestRunner:
  """A TestRunner instance is created for each worker thread."""

  # A method for each recorded page.
  def page1(self):
    """GET / (request 101)."""
    result = request101.GET('/')

    return result

  def __call__(self):
    """Called for every run performed by the worker thread."""
    self.page1()      # GET / (request 101)
```

```
    grinder.sleep(34)

# Instrument page methods.
Test(100, 'Page 1').record(TestRunner.page1)
```

Genereller Aufbau der Skripte:

1. Definieren der Klasse `TestRunner`, welche genau diesen Namen besitzen muss.
2. Die definierte Klasse `TestRunner` muss ausführbar sein. Dafür muss die Klasse die Methode `__call__` enthalten.
3. Das Importieren des Objektes grinder, mit dem Befehl `from net.grinder.script.Grinder`, das von der Engine zur Verfügung gestellt wird. Das grinder Objekt liefert Kontextinformationen wie Thread IDs, Logging und Statistiken.
4. Die Jython Skript Dateien müssen die Endung `.py` haben.

4.3.3.1 Script Recorder

Alternativ zum Erstellen der Skripte mit Jython ist es möglich Skripte mit dem Recorder aufzunehmen und abzuspielen. Der „TCPProxy" wird über das `startProxy` Shell-Skript ausgeführt. In der Konsole wird ausgegeben auf welche Adresse und Port der Proxy für die Aufnahme lauscht. In den Internetoptionen oder direkt im Browser muss der Proxy dementsprechend eingestellt werden. Während die GUI des „TCPProxy" geöffnet ist, findet die Aufnahme der HTTP Requests statt. Ist die Simulation des Szenerios fertig, klickt man auf den „Stop" Button.

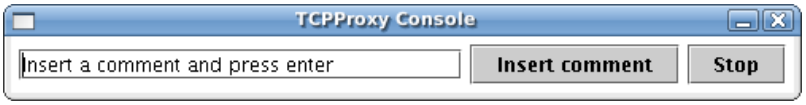

Abbildung 24: „TCPProxy" Oberfläche

War die Aufzeichnung erfolgreich, wird die Datei `grinder.py` im Ordner `bin` der grinder Installation erstellt oder falls diese bereits existieren sollte, überschrieben.

4.3.3.2 Datenbanken

Über eine JDBC Verbindung lassen sich Datenbanken stresstesten. Ein Beispiel Skript der Dokumentation für das Stresstesten einer Oracle Datenbank [72]:

```
# Some simple database playing with JDBC.
#
# To run this, set the Oracle login details appropriately and add the
# Oracle thin driver classes to your CLASSPATH.

from java.sql import DriverManager
from net.grinder.script.Grinder import grinder
from net.grinder.script import Test
from oracle.jdbc import OracleDriver

test1 = Test(1, "Database insert")
test2 = Test(2, "Database query")

# Load the Oracle JDBC driver.
DriverManager.registerDriver(OracleDriver())

def getConnection():
    return DriverManager.getConnection(
        "jdbc:oracle:thin:@127.0.0.1:1521:mysid", "wls", "wls")

def ensureClosed(object):
    try: object.close()
    except: pass

# One time initialisation that cleans out old data.
connection = getConnection()
statement = connection.createStatement()

try: statement.execute("drop table grinder_fun")
except: pass

statement.execute("create table grinder_fun(thread number, run
number)")

ensureClosed(statement)
ensureClosed(connection)

class TestRunner:
    def __call__(self):
        connection = None
        insertStatement = None
        queryStatement = None

        try:
            connection = getConnection()
            insertStatement = connection.createStatement()

            test1.record(insertStatement)
            insertStatement.execute("insert into grinder_fun
values(%d, %d)" %
                                    (grinder.threadNumber,
grinder.runNumber))

            test2.record(queryStatement)
            queryStatement.execute("select * from grinder_fun where
thread=%d" %
                                   grinder.threadNumber)

        finally:
            ensureClosed(insertStatement)
            ensureClosed(queryStatement)
            ensureClosed(connection)
```

4.3.3.3 Barriers

Mit Hilfe der Barrieres lassen sich einzelne Worker Threads synchronisieren, indem im Skript ein Punkt festgelegt wird. Einzelne Threads, die an diesem Punkt angekommen sind, warten bis alle anderen Threads auch angekommen sind. Barrieres werden nicht zwischen verschiedenen Thread Prozessen geteilt. Der Barrier-Punkt wird eingerichtet, indem alle Worker Threads diesen mit denselben Namen in der `__init__` Methode definiert haben [73]:

```
from net.grinder.script.Grinder import grinder

class TestRunner:
  def __init__(self):
    # Each worker thread joins the barrier.
    self.phase1CompleteBarrier = grinder.barrier("Phase 1")

  def __call__(self):
    # ... Phase 1 actions.
    # Wait for all worker threads to reach this point before
proceeding.
    self.phase1CompleteBarrier.await()
    # ... Further actions.
```

Barriers können für die Organisation des Skriptes nützlich sein. Beispielsweise könnten einzelne Tests innerhalb eines Szenarios voneinander getrennt werden, sodass diese sich nicht gegenseitig beeinflussen und die Ergebnisse verfälschen.

4.3.3.4 Verschiedene Nutzerprofile

Mit Hilfe der Einbindung einer Textdatei, die verschiedene Nutzerangaben beinhaltet, können verschiedene Nutzerprofile für einen Test verwendet werden. Zum Beispiel lassen sich in der Textdatei Nutzernamen und Passwörter hinterlegen. Für die Begründung der Nutzung von Textdateien statt Arrays wird die Limitierung von Jython/Java genannt. Für Arrays sei nur eine Größe von maximal 64KB möglich. Nachfolgend ein Beispielskript zur Einbindung einer Textdatei aus der Grinder Dokumentation [74]:

```
#
# testRandomise.py
#
import random
import string

class TestRandomise:
  def __init__(self, filename):
    self._users = []
    infile = open(filename, "r")
```

```
    for line in infile.readlines():
      self._users.append(string.split((line),','))
    infile.close()

  def getUserInfo(self):
    "Pick a random (user, password) from the list."
    return random.choice(self._users)


#
# Test script. Originally recorded by the TCPProxy.
#
from testRandomise import TestRandomise
tre = TestRandomise("users.txt")

class TestRunner:
    def __call__(self):
        # Get user for this run.
        (user, passwd) = tre.getUserInfo()

  # ...

  # Use the user details to log in.

        tests[2002].POST('https://host:443/securityservlet',
          ( NVPair('functionname', 'Login'),
            NVPair('pagename', 'Login'),
            NVPair('ms_emailAddress', user),
            NVPair('ms_password', passwd), ))
```

4.3.3.5 Assertions

Ergebnisse eines Tests können während dem Testablauf überprüft werden, genannt Assertion. Diese Funktion ist nur dadurch möglich, dass die Skripte vollen Zugriff auf die Testergebnisse haben. Mit Hilfe der Assertions können die Antworten vom Server auf ihre Richtigkeit überprüft werden. Um eine Assertion verwenden zu können, muss im Skript angegeben werden, dass die einzelnen Requests verzögert werden [75]:

```
grinder.statistics.delayReports = 1
request = HTTPRequest()
tests[testId].record(request)

log("Sending request %s " % requestString)
result = request.GET(requestString)
```

Als nächstes kann das Ergebnis auf dessen Inhalt überprüft werden. Sollte das Ergebnis den Erwartungen entsprechen, in folgendem Beispiel ob der Request nicht erfolgreich war, wird es anschließend in eine Datei geschrieben:

```
if string.find(result.getText(), "SUCCESS") < 1:
    grinder.statistics.forLastTest.setSuccess(0)
    writeToFile(result.getText())

# Write the response
def writeToFile(text):
    filename = "%s-page-%d.html" % (grinder.processName,
                                    grinder.runNumber)
    file = open(filename, "w")
    print >> file, text
    file.close()
```

Mit diesem Vorgehen lassen sich zum Beispiel zu einem späteren Zeitpunkt bei auftretenden Fehlern anhand der Ergebnisse mögliche Fehlerquellen analysieren. Bei der Überprüfung sollte darauf geachtet werden, dass nicht zu viel verarbeitet wird, sonst werden die Testergebnisse verfälscht. [75]

4.3.3.6 Verwenden von externen Daten

Grinder ermöglicht es externe Daten für Tests zu verwenden. Es ist möglich aus Textdateien oder Datenbanken zu lesen. Ein Code-Ausschnitt aus der Dokumentation zeigt, wie Daten aus einer Textdatei geladen werden und anschließend eine Request URL entsprechend angepasst wird [75]:

```
tests = {
    "News01"      : Test(1, "News 1 posting"),
    "Sport01"     : Test(2, "Sport 1 posting"),
    "Sport02"     : Test(3, "Sport 2 posting"),
    "Trading01"   : Test(4, "Trading 1 query"),
    "LifeStyle01" : Test(5, "LifeStyle 1 posting"),
}

testId = random.choice(tests.keys())

log("Reading XML file %s " % testId)

file = open("./CAAssets/"+testId+".xml", 'r')
fileStr = URLEncoder.encode(String(file.read()))
file.close()

# Send the request to the server
requestString = "%s%s%s%s" % (SERVER, URI, "?xmldata=", fileStr)
requestString = string.join(requestString.split(), "")

grinder.statistics.delayReports = 1
request = HTTPRequest()
tests[testId].record(request)

log("Sending request %s " % requestString)
result = request.GET(requestString)
```

Das verwenden von externen Daten über eine Datenbank erfolgt über JDBC.

4.3.3.7 Weitere Features

Es wird nicht auf alle Features einzeln eingegangen, da es sich häufig nur um die Anwendung von neuen Methoden handelt. Eine Liste weiterer nützlichen Features von Grinder [76]:

1. HTTP digest authentication
2. HTTP Cookies
3. HTTP multipart form submission
4. Enterprise Java Beans
5. HTTP Web Service
6. JAX-RPC Web Serivce
7. XML-RPC Web Service
8. Email
9. Unterschiedliche Last Verteilung für verschiedene Szenarien

4.3.4 Konfigurieren der virtuellen Nutzer

Die Konfiguration der Anzahl virtueller Nutzer, die die Anwendung testen, wird in der `grinder.properties` Datei angegeben unter dem Ordner `grinder-agent`.

Aus den Angaben, die man in den `grinder.properties` eingestellt hat, lassen sich die virtuellen Nutzer mit folgender Formel berechnen [77]:

```
number of worker threads x number of worker processes x number test
machines
```

Die Datei zur Konfigurierung der Generierung von virtuellen Nutzer:

```
#################################################################
###################
# The number of worker processes the agent should start.
# Example: grinder.processes=1
#################################################################
###################
grinder.processes=1

#################################################################
###################
# The number of worker threads that each worker process spawns.
# Example:grinder.threads=1
```

```
#########################################################################
####################
grinder.threads=1

#########################################################################
####################
# The number of runs of the test script each thread performs.
# 0 means "run forever", and should be used when you are using the
console to control
# your test runs.
# Example: grinder.runs=0
#########################################################################
####################
grinder.runs=0
```

4.3.5 Distributed Testing

Distributed Testing kann umgesetzt werden, indem eine relative Gewichtung der im Skript definierten Tests, angegeben wird. Eine beispielhafte Umsetzung der definierten Tests sieht wie folgt aus [78]:

```
def doCREATEtest():
    print 'Doing CREATE test ...'
def doREADtest():
    print 'Doing READ   test ...'
def doUPDATEtest():
    print 'Doing UPDATE test ...'
def doDELETEtest():
    print 'Doing DELETE test ...'
```

Und die dazugehörige Verteilung der Tests wie folgt:

```
g_Weights = {
    'CREATE': 2,
    'READ'  : 4,
    'UPDATE': 3,
    'DELETE': 1,
}
```

4.3.6 Ausführung der Tests

Um die Skripte ausführen zu können muss die Console laufen und das Shell Skript `startAgent.sh` gestartet werden, das sich automatisch mit der Console, der GUI, verbindet und kommuniziert. Haben sich Console und Agent erfolgreich verbunden, wird in der GUI der Button klickbar für die Ausführung der Tests. Während der Ausführung können im Tab „Graphs“ und „Results“ die Statistiken beobachtet werden.

4.3.7 Konsolenparameter

Eine wichtige Fähigkeit für remotes verwenden von Grinder ist die Möglichkeit Tests ohne GUI ausführen zu können. Dafür muss die „Console" mit dem Parameter `–headless` ausgeführt werden. Der komplette Konsolenbefehl [79]:

```
java -classpath lib/grinder.jar net.grinder.Console –headless
```

Über die Konsole können die Tests gestartet, einige Konfigurationen vorgenommen, die Testergebnisse ausgegeben werden, und auch zur Speicherortfestlegung der Ergebnisse.

4.3.8 Remote Testing

Für Remote Testing muss lediglich in der Konfiguration des Grinder Agenten die IP-Adresse der Grinder Console angegeben und auf der Injection Maschine ausgeführt werden. Wurde eine korrekte Konfiguration angelegt und der Agent ausgeführt, erscheint folgende Ausgabe in der Konsole [80]:

```
2016-08-14 00:34:29,486 INFO  agent: The Grinder 3.11
2016-08-14 00:34:29,662 INFO  agent: connected to console at /192.168.2.103:6372
2016-08-14 00:34:29,662 INFO  agent: waiting for console signal
```

4.3.9 Auswertung der Ergebnisse

Im „Graphs“ und „Results“ Tab werden die Ergebnisse der Tests dargestellt. Im “Graphs” Tab wird eine kleine Übersicht der Ergebnisse angezeigt. Die Graphen zeigen die letzten 25 TPS (Transactions per Second) eines Tests innerhalb einer Sampleperiode an. Die Höhe des Graphen wird mit den “peak”, also höchste, TPS ermittelt. Die Farben, die im Graphen verwendet werden, sind abhängig von der ermittelten Reaktionszeit. Lange Reaktionszeiten werden mit rot und kurze mit gelb angegeben.

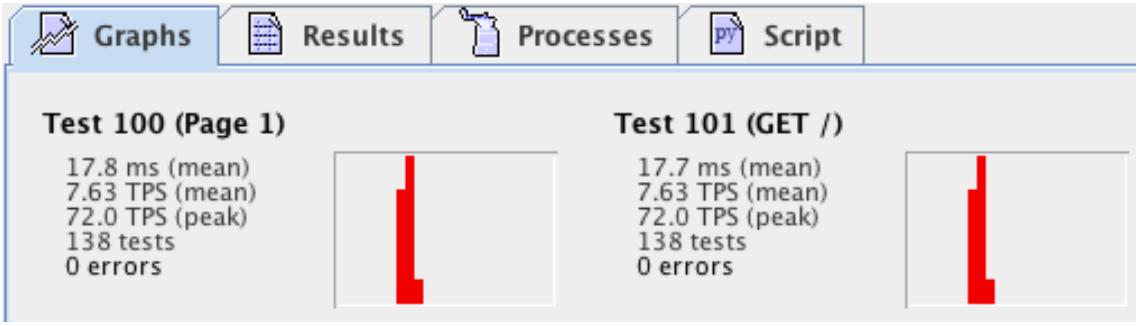

Abbildung 25: Graphen zu jedem definierten Test

Die Tabelle des „Results“ Tab wir durch folgende Spalten definiert [81]:

1. **Test**: Die Testnummer, die im Skript definiert wurde.
2. **Description**: Die Testbeschreibung, die im Skript angegeben wurde.
3. **Errors**: Die gesamte Anzahl von Testdurchläufen, die fehlschlugen.
4. **Mean Time**: Durchschnittliche Zeit, die benötigt wir den Test auszuführen einschließlich empfangen der Server Antwort, in Millisekunden.
5. **Mean Time Standard**: Die Normalabweichung der durchschnittlich benötigten Zeit, in Millisekunden.
6. **TPS**: Transactions per Second, die Anzahl Wiederholungen des Tests innerhalb eines Ein-Sekunden-Intervalls.
7. **Peak TPS**: Höchste maximale Anzahl ausgeführter Transaktionen innerhalb eines Ein-Sekunden-Intervalls.

Weitere Spalten die durch das HTTP Plugin hinzugefügt werden:

1. **Mean Response Length**: Die durchschnittliche Größe einer Serverantwort, in Bytes.
2. **Response Bytes per Second**: Die durchschnittliche Anzahl von Bytes, die innerhalb einer Sekunde vom Server erhalten werden. Gibt Auskunft darüber wie viel

Bandbreite die Anwendung benötigt. Dies ermöglicht keine Aussage über den gesamt benötigten Traffic.

3. **Response Error**: Die gesamte Anzahl HTTP Fehler (404, 405, etc.) die während dem Test auftraten.
4. **Mean Time to Resolve Host**: Die durchschnittlich benötigte Zeit um die IP Adresse des Servers aufzulösen, in Millisekunden.
5. **Mean Time to Establish Connection**: Die durchschnittlich benötigte Zeit um eine TCP Verbindung mit dem Server herzustellen, in Millisekunden.
6. **Mean Time to First Byte**: Die durchschnittlich benötigte Zeit bis zum Erhalt der ersten Bytes vom Server, in Millisekunden.

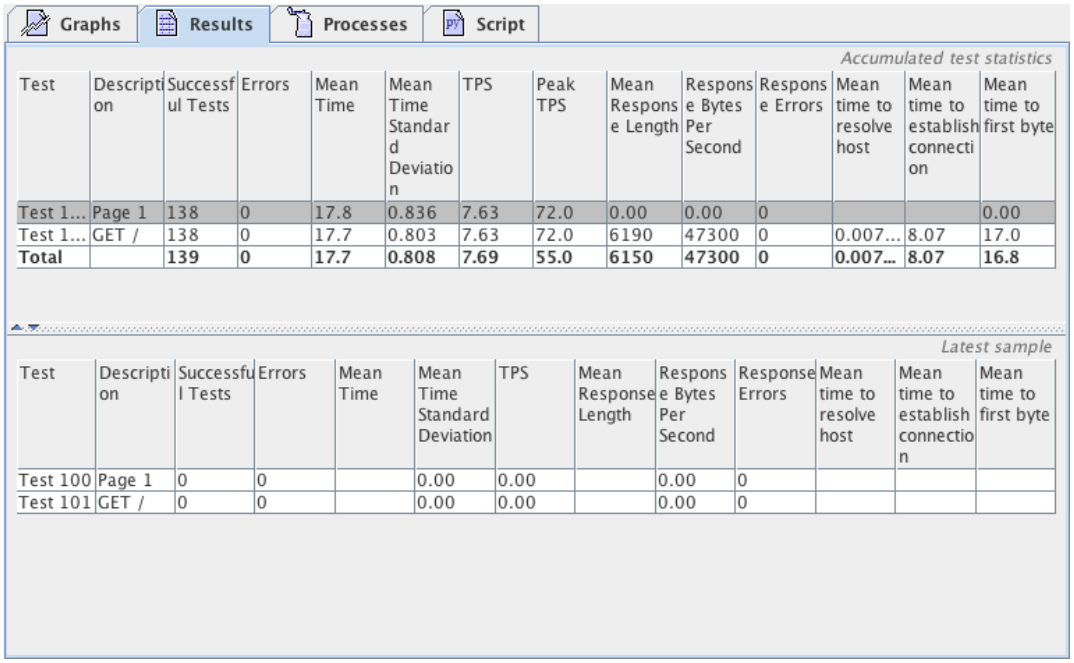

Abbildung 26: Tabelle mit Ergebnissen zu jedem Test

4.3.10 Realtime Monitoring

Für Grinder wird das Tool Graphite, wie bei Gatling und JMeter auch, für Echtzeitüberwachung aufgeführt. Damit Grinder mit Graphite kommunizieren kann, wird Grinder2Graphite benötigt. Bei Grinder2Graphite handelt es sich um ein externes Tool. Darum wird in dieser Arbeit auf die Umsetzung dieses Tools nicht näher eingegangen. [82]

4.3.11 HTTP/2 Unterstützung

Von einer Unterstützung für das HTTP/2 Protokoll ist nicht auszugehen, da die letzte Aktualisierung von Grinder 20.12.2012 stattfand und das HTTP/2 Protokoll erst seit Mai 2015 spezifiziert ist. [83]

4.3.12 WebSocket Verbindungen

Bislang existiert keine Unterstützung für WebSocketverbindungen. [84]

5 Performance-Test-Plan am Beispiel computer-database.gatling.io

5.1 Non Functional Requirements

Für das Evaluieren der einzelnen Performance Frameworks gibt es für die Anwendung keine besonderen vorher festzulegenden Anforderungen. Somit können wir eine beliebige Anwendung testen. Deshalb wird die von Gatling bereitgestellte Anwendung „computer-database.gatling.io" in den verschiedenen Tests zum Einsatz kommen. Sollte beim Evaluieren auffallen, dass eine Funktion des Frameworks auf Grund einer fehlenden Deckung durch die Anwendung nicht durchführbar ist, wird nach einer anderen Anwendung Ausschau gehalten, die die nötige Funktion besitzt. Bevor mit dem Testen angefangen werden kann, müssen die Use-Cases identifiziert und festgelegt werden.

Folgende Key-Use-Cases wurden in der Dokumentation von Gatling festgelegt [85]:

1. Der Nutzer landet auf der Anwendung
2. Der Nutzer sucht beispielsweise nach „Macbook"
3. Der Nutzer öffnet eines der für ihn relevanten Modelle
4. Der Nutzer geht zurück zur Startseite
5. Der Nutzer blättert durch die verschiedenen Seiten
6. Der Nutzer erstellt ein neues Modell

5.1.1 Test Environment Build

Für das Evaluieren eines Frameworks ist die Testumgebung nicht weiter zu beachten, da die Performance der Anwendung nicht im Fokus der Arbeit steht. Jedoch ist es relevant, welche Möglichkeiten das Framework bietet, um die Tests von außerhalb ausführen zu lassen. Außerdem muss die Anwendung nicht mit Daten bestückt werden, da dies bereits durch Nutzer der Anwendung, geschehen ist. Es ist lediglich zu beachten, dass das Framework richtig konfiguriert ist, damit die Funktionen korrekt evaluiert werden können.

5.2 Use-Case-Scripting

Anhand der definierten Use-Cases werden die Use-Case-Skripte erstellt. Die Frameworks verfolgen beim Erstellen der Skripte verschiedene Ansätze. Gatling, JMeter und Grinder bieten einen Recorder an, der dazu dient die Aktionen im Browser aufzuzeichnen und in ein Skript umzuwandeln bzw. in JMeter die einzelnen Sampler-Elemente umzuwandeln. In Gatling und Grinder können die Skripte weiter auf die Bedürfnisse angepasst werden. In JMeter müssen dem Testplan weitere Elemente hinzugefügt werden, um das Verhalten des Tests anzupassen.

Checkpoints sind in dieser Arbeit nicht nötig, da von einer fertigen Anwendung ausgegangen wird, deren Fehler bereits behoben worden sind. Jedoch wären folgende Checkpoints möglich:

1. Nach jeder Aktion.
2. Nach dem „post" Befehl, um die Richtigkeit der Transaktion der Datenbank zu prüfen.

5.3 Performance Test Scenario Build

In Bezug auf die Arbeit, dem Evaluieren der Frameworks, muss überprüft werden, welche Testarten das Framework ermöglicht und wie fein granular die Skripte im Allgemeinen erstellt werden können. Es sollte überprüft werden, ob das Framework bei verschieden eingestellten Testzeiten korrekt funktioniert und nicht das System abstürzen lässt.

5.4 Performance Test Execution

Die erstellten Tests müssen ausgeführt werden, um kontrollieren zu können, dass das Framework in der Lage ist diese korrekt auszuführen.

5.5 Post-Test Analysis and Reporting

Sobald die Tests ausgeführt worden sind, müssen die gesammelten Daten ausgewertet werden. Hier ist zu überprüfen, welche Möglichkeiten die verschiedenen Frameworks bieten, um ihre Ergebnisse darzustellen. Relevant ist zu wissen, ob für die grafische Dar-

stellung der Ergebnisse zusätzliche Schritte unternommen werden müssen oder ob die Ergebnisse schon in einer auswertbaren Form ausgegeben werden. Ein weiterer Punkt ist, ob die Ausgabe der Ergebnisse mit den eigenen Bedürfnissen erweiterbar ist.

6 Vergleich der Test-Ausführung der Tools

Im letzten Abschnitt der Arbeit werden die Testergebnisse der einzelnen Frameworks eines Szenarios gegenübergestellt und verglichen. Es wird die Komplexität des Ablaufs, die durch die verschiedenen Frameworks gegeben ist, bis zur Fertigstellung eines Testplans verglichen. Für die geplanten Tests werden die in Kapitel 5.1 beschriebenen Use-Cases verwendet. Das Use-Case-Skripting wird mit Hilfe der HTTP Proxy Recorder des jeweiligen Frameworks umgesetzt. Im Rahmen dieser Arbeit ist es nicht möglich alle Features der Frameworks anzuwenden.

6.1 Gatling

Es wird der Recorder von Gatling gestartet, der das Szenario aufzeichnet in dem mit der Anwendung im Browser interagiert wird. Dabei wird folgendes Scala-Skript erstellt:

```
package bachelor

import scala.concurrent.duration._

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.jdbc.Predef._

class computerSimulation extends Simulation {

   val httpProtocol = http
      .baseURL("http://computer-database.gatling.io")
      .inferHtmlResources()

   .acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.
9,*/*;q=0.8")
      .acceptLanguageHeader("en-US,en;q=0.5")
      .userAgentHeader("Mozilla/5.0 (Macintosh; Intel Mac OS X 10.11;
rv:47.0) Gecko/20100101 Firefox/47.0")

   val headers_0 = Map("Accept-Encoding" -> "gzip, deflate")

   val headers_1 = Map(
      "Accept" -> "*/*",
      "Accept-Encoding" -> "gzip, deflate")

   val headers_8 = Map("Range" -> "bytes=10200000-10499999")

    val uri1 =
"http://download.cdn.mozilla.net/pub/firefox/releases/48.0/update/mac/
en-US/firefox-47.0-48.0.partial.mar"

   val scn = scenario("computerSimulation")
```

```
      .exec(http("request_0")
         .get("/")
         .headers(headers_0)
         .resources(http("request_1")
         .get("/favicon.ico")
         .headers(headers_1)
         .check(status.is(404)),
            http("request_2")
         .get("/favicon.ico")
         .headers(headers_0)
         .check(status.is(404))))
      .pause(4)
      .exec(http("request_3")
         .get("/computers?f=macbook")
         .headers(headers_0))
      .pause(2)
      .exec(http("request_4")
         .get("/computers/6")
         .headers(headers_0))
      .pause(5)
      .exec(http("request_5")
         .get("/computers")
         .headers(headers_0))
      .pause(2)
      .exec(http("request_6")
         .get("/computers?p=1")
         .headers(headers_0))
      .pause(2)
      .exec(http("request_7")
         .get("/computers?p=2")
         .headers(headers_0)
         .resources(http("request_8")
         .get(uri1 + "")
         .headers(headers_8)))
      .pause(1)
      .exec(http("request_9")
         .get("/computers?p=3")
         .headers(headers_0))
      .pause(1)
      .exec(http("request_10")
         .get("/computers?p=4")
         .headers(headers_0))
      .pause(1)
      .exec(http("request_11")
         .get("/computers?p=5")
         .headers(headers_0)
         .resources(http("request_12")
         .get("/computers?p=6")
         .headers(headers_0),
            http("request_13")
         .get("/computers?p=7")
         .headers(headers_0)))
      .pause(1)
      .exec(http("request_14")
         .get("/computers/new")
         .headers(headers_0))
      .pause(12)
      .exec(http("request_15")
         .post("/computers")
         .headers(headers_0)
         .formParam("name", "Test Macbook")
         .formParam("introduced", "")
```

```
          .formParam("discontinued", "")
          .formParam("company", "1"))

   setUp(scn.inject(rampUsers(10) over(10
seconds))).protocols(httpProtocol)
}
```

Zu sehen sind die einzelnen aufgenommenen Requests mit den jeweils dazwischenliegenden Wartezeiten. Die selben Wartezeiten werden bei den restlichen Frameworks auch zum Einsatz kommen.

STATISTICS — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Req/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	180	180	0	0%	4.39	25	37	40	59	101	1074	45	77
request_0	10	10	0	0%	0.244	33	38	45	79	95	100	45	19
request_...direct 1	10	10	0	0%	0.244	35	36	44	53	57	59	40	7
request_...direct 1	10	10	0	0%	0.244	34	35	37	61	72	75	40	12
request_...direct 1	10	10	0	0%	0.244	33	35	37	61	72	75	40	11
request_3	10	10	0	0%	0.244	35	39	43	62	73	76	43	11
request_4	10	10	0	0%	0.244	34	37	40	50	54	55	39	6
request_5	10	10	0	0%	0.244	35	40	40	43	43	44	39	2
request_6	10	10	0	0%	0.244	34	36	39	42	42	43	37	2
request_7	10	10	0	0%	0.244	35	39	41	68	85	90	43	15
request_8	10	10	0	0%	0.244	25	33	52	85	104	109	44	24
request_9	10	10	0	0%	0.244	35	38	46	623	983	1074	145	309
request_10	10	10	0	0%	0.244	34	36	38	50	50	51	38	5
request_11	10	10	0	0%	0.244	35	36	37	39	40	41	36	1
request_13	10	10	0	0%	0.244	35	36	37	40	42	43	36	2
request_12	10	10	0	0%	0.244	34	36	37	40	42	43	36	2
request_14	10	10	0	0%	0.244	33	35	35	37	38	39	35	1
request_15	10	10	0	0%	0.244	34	36	38	40	40	41	36	2
request_...direct 1	10	10	0	0%	0.244	36	37	38	51	58	60	39	6

Abbildung 27: Gatling Statistiktabelle

6.2 JMeter

Auch bei JMeter wird der Script Recorder für das Aufzeichnen der einzelnen Requests verwendet. Es werden aber keine Wartezeiten zwischen den Requests aufgenommen. Diese müssen nachträglich mit dem Constant Timer, wie in diesem Beispiel, hinzugefügt werden. Es werden dieselben Wartezeiten wie die im aufgezeichneten Skript von Gatling verwendet.

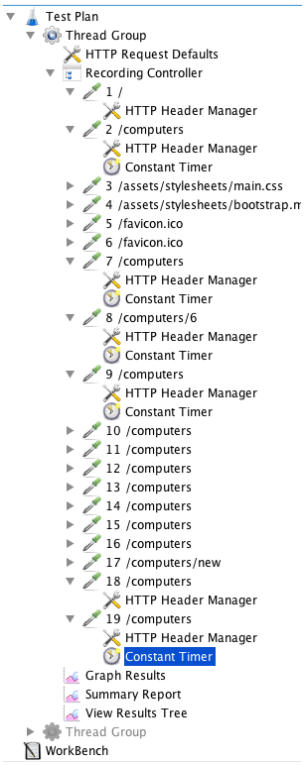

Abbildung 28: HTTP Sampler mit Constant Timer in deren Scope

Für die anschließende Erstellung der Ergebnisgrafiken benutzen wir den in Kapitel 4.2.6 angegeben Konsolenbefehl um den Dashboard Report zu erstellen.

Label	#Samples	KO	Error %	90th pct	95th pct	99th pct	Throughput	KB/sec	Min	Max
Total	**150**	**0**	**0.00%**	**68.00**	**71.00**	**150.86**	**3.66**	**25.90**	**33**	**158**
1 /	10	0	0.00%	149.40	158.00	158.00	1.11	8.16	68	158
2 /computers	10	0	0.00%	39.00	39.00	39.00	1.12	8.11	34	39
3 /computers	10	0	0.00%	43.60	44.00	44.00	1.12	8.43	35	44
4 /computers/6	10	0	0.00%	38.00	38.00	38.00	1.12	6.16	33	38
5 /computers	10	0	0.00%	38.70	39.00	39.00	1.12	8.11	34	39
6 /computers	10	0	0.00%	37.90	38.00	38.00	1.12	8.17	34	38
7 /computers	10	0	0.00%	38.80	39.00	39.00	1.12	8.26	33	39
8 /computers	10	0	0.00%	36.90	37.00	37.00	1.12	8.25	33	37
9 /computers	10	0	0.00%	37.90	38.00	38.00	1.12	8.15	34	38
10 /computers	10	0	0.00%	35.90	36.00	36.00	1.12	8.16	33	36
11 /computers	10	0	0.00%	36.80	37.00	37.00	1.12	8.21	33	37
12 /computers	10	0	0.00%	38.70	39.00	39.00	1.12	8.17	33	39
13 /computers/new	10	0	0.00%	39.80	40.00	40.00	1.12	5.92	34	40
14 /computers	10	0	0.00%	73.80	74.00	74.00	1.12	8.33	67	74
15 /computers	10	0	0.00%	133.40	144.00	144.00	1.12	8.11	34	144

Abbildung 29: Dashboard Report Tabelle

6.3 Grinder

Grinder besitzt auch einen Script Recorder, der für die Aufzeichnung der Use-Cases verwendet wird. Es wird die Grinder „Console“ und anschließend der „TCPProxy“ über das Terminal gestartet, wobei die Proxy Einstellungen so eingestellt werden müssen, wie von Grinder angezeigt. Läuft der „TCPProxy“, kann mit der Anwendung im Browser interagiert werden. Nach der Szenario Durchführung erstellt Grinder das Jython Skript. In dem Skript sind bereits die Wartezeiten integriert, angegeben in Millisekunden. Diese Wartezeiten werden mit denen Zeiten aus den anderen Skripten abgestimmt. Ein Ausschnitt des generierten Skriptes:

```
def __call__(self):
  """Called for every run performed by the worker thread."""
  self.page1()      # GET / (request 101)

  grinder.sleep(38)
  self.page2()      # GET computers (request 201)

  grinder.sleep(4000)
  self.page3()      # GET computers (request 301)

  grinder.sleep(2000)
  self.page4()      # GET 6 (request 401)

  grinder.sleep(5000)
  self.page5()      # GET computers (request 501)

  grinder.sleep(2000)
  self.page6()      # GET computers (request 601)

  grinder.sleep(2000)
  self.page7()      # GET computers (request 701)

  grinder.sleep(1000)
  self.page8()      # GET computers (request 801)

  grinder.sleep(1000)
  self.page9()      # GET computers (request 901)

  grinder.sleep(1000)
  self.page10()     # GET computers (request 1001)

  grinder.sleep(1000)
  self.page11()     # GET computers (request 1101)

  grinder.sleep(1000)
  self.page12()     # GET computers (request 1201)

  grinder.sleep(1000)
  self.page13()     # GET new (request 1301)

  grinder.sleep(12000)
  self.page14()     # POST computers (request 1401)
```

```
    grinder.sleep(32)
    self.page15()     # GET computers (request 1501)


# Instrument page methods.
Test(100, 'Page 1').record(TestRunner.page1)
Test(200, 'Page 2').record(TestRunner.page2)
Test(300, 'Page 3').record(TestRunner.page3)
Test(400, 'Page 4').record(TestRunner.page4)
Test(500, 'Page 5').record(TestRunner.page5)
Test(600, 'Page 6').record(TestRunner.page6)
Test(700, 'Page 7').record(TestRunner.page7)
Test(800, 'Page 8').record(TestRunner.page8)
Test(900, 'Page 9').record(TestRunner.page9)
Test(1000, 'Page 10').record(TestRunner.page10)
Test(1100, 'Page 11').record(TestRunner.page11)
Test(1200, 'Page 12').record(TestRunner.page12)
Test(1300, 'Page 13').record(TestRunner.page13)
Test(1400, 'Page 14').record(TestRunner.page14)
Test(1500, 'Page 15').record(TestRunner.page15)
```

Ist der Test beendet, findet man im „Results" Tab eine Tabelle mit den Ergebnissen aller Tests. Die Ergebnisse werden für eine spätere Auswertung außerdem in einer Datei gespeichert.

Accumulated test statistics

Test	Description	Successful Tests	Errors	Mean Time	Mean Time Standard Deviation	TPS	Peak TPS	Mean Response Length	Response Bytes Per Second	Response Errors	Mean time to resolve host	Mean time to establish connection	Mean time to first byte
Test 101	GET /	10	0	494	1.94	0.277	10.0	0.00	0.00	0	440	460	488
Test 201	GET com...	10	0	39.9	2.12	0.277	10.0	7270	2010	0	0.00	15.3	38.6
Test 301	GET com...	10	0	36.7	1.19	0.277	10.0	7570	2100	0	0.00	15.1	35.8
Test 401	GET 6	10	0	34.8	1.17	0.277	10.0	5490	1520	0	0.100	15.5	34.2
Test 501	GET com...	10	0	35.5	1.57	0.277	10.0	7270	2010	0	0.00	15.3	34.3
Test 601	GET com...	10	0	34.7	1.10	0.277	10.0	7320	2030	0	0.00	15.4	33.9
Test 701	GET com...	10	0	35.1	1.14	0.277	9.00	7410	2050	0	0.00	15.6	34.1
Test 801	GET com...	10	0	34.5	0.922	0.277	8.00	7400	2050	0	0.00	15.0	33.8
Test 901	GET com...	10	0	34.6	0.800	0.277	6.00	7310	2030	0	0.00	14.9	33.6
Test 1001	GET com...	10	0	35.8	1.54	0.277	6.00	7310	2030	0	0.100	15.3	34.6
Test 1101	GET com...	10	0	34.8	1.33	0.277	5.00	7360	2040	0	0.00	15.3	33.8
Test 1201	GET com...	10	0	39.4	8.28	0.277	5.00	7330	2030	0	0.00	18.3	37.3
Test 1301	GET new	10	0	35.5	1.86	0.277	5.00	5280	1460	0	0.100	15.4	34.9
Test 1401	POST co...	10	0	251	1.73	0.277	8.00	0.00	0.00	0	0.100	15.2	251
Test 1501	GET com...	10	0	35.0	0.894	0.277	9.00	7400	2050	0	0.100	14.9	34.0
Total		**150**	**0**	**80.8**	**123**	**4.16**	**20.0**	**6110**	**25400**	**0**	**29.4**	**45.1**	**79.4**

Abbildung 30: "Results" Tab in der Grinder GUI mit Ergebnisse

7 Bewertungen der Frameworks

7.1 Gatling Bewertung

Die Stärke von Gatling ist der überschaubare Programmiercode. Positiv fällt die Simulierung von Nutzerprofilen, die nahe an der Realität liegen auf. Diese lassen sich durch den mitgelieferten Recorder aufzeichnen und anschließend die generierten Skripte genauer anpassen. Jedes der Profile erzeugt dabei eine eigene Verbindung mit dem Server. Versucht man, mit Hilfe von vielen Tests mit einer sehr kurzen Lebensspanne, große Last auf dem Server zu erzeugen, kann es dazu führen, dass der Last erzeugende Rechner langsamer wird oder sogar abstürzt. In diesen Fällen sollte das Besprochene „Scaling Out" betrieben werden. Des Weiteren sollte man sein Injection-Model überdenken, ob vielleicht weniger gleichzeitige User ausreichend sind für die zu testende Anwendung. Außerdem kann Gatling so konfiguriert werden, dass mehrere Nutzer die gleiche HTTP Verbindung verwenden und unter sich teilen. [86] Außerdem fällt auch die Möglichkeit zur Darstellung der Daten, mögliches einbinden der Testskripte in eine Versionisierung und GUI-lose Bedienbarkeit, positiv auf. Neutral aufgefallen hingegen ist, dass die Umsetzung der Skalierbarkeit mit Aufwand verbunden ist.

7.2 JMeter Bewertung

JMeter lässt sich auf allen Systemen ausführen, die Java unterstützen. Des Weiteren lässt sich die Last unkompliziert und schnell durch JMeter Server erweitern. Testpläne lassen sich zusammenstellen ohne Kenntnisse über eine Programmiersprache haben zu müssen. Die Auswertung der Daten geschieht unkompliziert und problemlos durch Hinzufügen der Listener, jedoch sind die resultierenden Tabellen schlecht beschriftet und nicht sofort verständlich. Möchte man die JMeter Dashboard Graphen erstellen, muss JMeter auf unkonventionellerweise eine Konfigurationsdatei angepasst werden und über die Konsole das Dashboard erstellt werden, was nicht dem generellen GUI Ansatz von JMeter folgt. Für fortgeschrittene Nutzer lässt sich JMeter außerdem komplett ohne GUI steuern, dies ist außerdem für Remote- oder Distributed Testing nötig.

7.3 Grinder Bewertung

Mit Grinder lassen sich schnell, mit Hilfe des „TCPProxy“, Tests automatisch erstellen und anschließend mit den Agenten ausführen. Die Einstellung der virtuellen Nutzer, die in einem Test ausgeführt werden sollen, gestaltet sich umständlich über die Anpassung der `grinder.properties`. Dabei bezieht sich die Konfigurationsdatei auf alle weiteren auszuführenden Testskripte. Des Weiteren ist die Aufbereitung der Ergebnisse veraltet und nicht mehr visuell ansprechend. Dafür können jedoch Plugins aushelfen, die die Ergebnisse moderner und leserlicher aufbereiten. Insgesamt erfüllt Grinder seinen Zweck und unterstützt alle wichtigen Features, um reichhaltige Tests zu erstellen. Auffallend ist das veraltete Oberflächendesign und die fehlende Unterstützung neuerer Protokolle wie WebSocket und HTTP/2.

8 Zusammenfassung und Ausblick

Der Vergleich der Frameworks verdeutlicht, dass größere Unterschiede der Frameworks nicht bestehen. Die Vorgehensweisen der Tools können im Detail verschieden sein, führen jedoch alle zu dem gleichen Ergebnis, dass sich mit ihnen reichhaltige komplexe Testpläne erstellen lassen. Auch größere Unterschiede in der Beschreibung der Ergebnisse sind nicht vorhanden, dafür bestehen Unterschiede im Vorgang der Erstellung und der grafischen Darstellung der Ergebnisse. Je moderner das vorliegende Tool ist, desto leichter lassen sich Diagramme erstellen und lesen, sowie auswerten. Auffällig sind die Unterschiede in der Gestaltung der grafischen Oberfläche. Es ist nicht zu übersehen, welche Tools bereits länger existieren als andere (Gatling und JMeter in Vergleich zu Grinder) und daher durch eine stetige Weiterentwicklung ein moderneres Design aufweisen. Die Unterstützung der neuen Protokolle HTTP/2 und WebSocket werden nur teilweise von den Frameworks gedeckt. Während Gatling und JMeter WebSocket unterstützen, ist dies bei Grinder nicht der Fall. HTTP/2 Unterstützung ist von allen drei Frameworks nicht gegeben. Gatling und JMeter haben die Unterstützung der fehlenden Protokolle aber in Planung.

Es ist fraglich, ob die hier vorgestellte Vorgehensweise in Zukunft weiter angewendet wird, da zu beobachten ist, dass in der Cloud Lösungen für das Performance Testing zu finden sind. Diese cloudbasierten Angebote sind zum Beispiel Loader.io, Blazemeter und GTmetrix. Der Vorteil dieser ist, dass man sich keine Gedanken über die Load Injektion machen muss. Es kann unkompliziert über einen Schieberegler die Anzahl der virtuellen Nutzer eingestellt werden, die die Anwendung testen sollen. Außerdem ist es nicht nötig sich mit der Installation der ausgewählten Tools, sowie der Anzahl nötiger Maschinen, auseinanderzusetzen. Diese Aufgaben übernimmt der Anbieter. Der Nachteil solcher Anbieter ist, dass ab einer bestimmten Anzahl virtueller Nutzer der Service nicht mehr kostenlos ist. Loader.io verlangt beispielweise ab 100.000 Nutzern 99$ im Monat. Der kostenlose Service beschränkt sich auf 10.000 Nutzer, ein „target host“, einen einminütigen Test und zwei URLs pro Test. Dies ist für die meisten Fälle nicht ausreichend, siehe Kapitel 6, um die Performance einer Webanwendung ausgiebig zu testen. [87]

9 Literaturverzeichnis

[1] I. Molyneaux, „Performance Measurement," in *The Art of Application Performance Testing, 2nd Edition*, O'Reilly Media, 2014, pp. 2-3.

[2] I. Molyneaux, „Performance Standards," in *The Art of Application Performance Testing, 2nd Edition*, O'Reilly Media, 2014, pp. 3-4.

[3] R. Arora, „What is pipe cleaning in software testing?," 16 5 2016. [Online]. URL: https://www.quora.com/What-is-pipe-cleaning-in-software-testing. [Zugriff am 29 7 2016].

[4] Tutorialspoint, „What is Volume Testing?," [Online]. URL: http://www.tutorialspoint.com/software_testing_dictionary/volume_testing.htm. [Zugriff am 20 7 2016].

[5] Tutorialspoint, „What is an Isolation Testing?," [Online]. URL: http://www.tutorialspoint.com/software_testing_dictionary/isolation_testing.htm. [Zugriff am 25 7 2016].

[6] Tutorialspoint, „What is Stress Testing?," [Online]. URL: http://www.tutorialspoint.com/software_testing_dictionary/stress_testing.htm. [Zugriff am 25 7 2016].

[7] Tutorialspoint, „What is Soak Testing?," [Online]. URL: http://www.tutorialspoint.com/software_testing_dictionary/soak_testing.htm. [Zugriff am 28 7 2016].

[8] Tutorialspoint, „What is Load Testing?," [Online]. URL: http://www.tutorialspoint.com/software_testing_dictionary/load_testing.htm. [Zugriff am 29 7 2016].

[9] Tutorialspoint, „Home," [Online]. URL: http://www.tutorialspoint.com/software_testing_dictionary/index.htm. [Zugriff am

25 7 2016].

[10] I. Molyneaux, „Performance Test Execution Checklist," in *The Art of Application Performance Testing, 2nd Edition*, pp. 225-231.

[11] Oracle, „ Oracle® Load Testing Load Testing User's Guide," 2010. [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#BGBICEFF. [Zugriff am 12 6 2016].

[12] Oracle, „Planning for Load Testing: Phases of Scalability Testing," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId1. [Zugriff am 22 6 2016].

[13] Oracle, „Planning for Load Testing: Criteria for Accurate Scalability Testing," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId2. [Zugriff am 24 6 2016].

[14] Oracle, „Planning for Load Testing: Determine Additional Tools to Perform Testing and Diagnosis," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId3. [Zugriff am 24 6 2016].

[15] Oracle, „Planning for Load Testing: Determining the Hardware Needed to Execute the Tests," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId4. [Zugriff am 24 6 2016].

[16] Oracle, „Planning for Load Testing: Who Should be Responsible for Load Testing?," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId5. [Zugriff am 24 6 2016].

[17] oracle, „Planning for Load Testing: What to Avoid When Testing for Scalability," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId6. [Zugriff am 25 6 2016].

[18] Oracle, „Planning for Load Testing: Performing Scalability Testing," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId7. [Zugriff am 25 6 2016].

[19] Oracle, „Performing Scalability Testing: Define the Process," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId8.

[20] Oracle, „Performing Scalability Testing: Define the Criteria," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId8. [Zugriff am 25 6 2016].

[21] Oracle, „Performing Scalability Testing: Planning the Scalability Tests," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId8. [Zugriff am 25 6 2016].

[22] Oracle, „Performing Scalability Testing: Planning the Load Test Scenarios," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId11. [Zugriff am 26 6 2016].

[23] Oracle, „Performing Scalability Testing: Create and Verify the Test Scripts," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId12. [Zugriff am 26 6 2016].

[24] Oracle, „Performing Scalability Testing: Create and Verify the Load Test Scenarios," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId13. [Zugriff am 26 6 2016].

[25] Oracle, „Performing Scalability Testing: Execute the Tests," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId14. [Zugriff am 26 6 2016].

[26] Oracle, „Performing Scalability Testing: Evaluate the Results," [Online]. URL:

https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId15. [Zugriff am 27 6 2016].

[27] Oracle, „Performing Scalability Testing: Generate Analysis Reports," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId16. [Zugriff am 27 6 2016].

[28] Gatling, „Gatling," [Online]. URL: http://gatling.io/docs/2.2.2/#gatling. [Zugriff am 20 7 2016].

[29] D. Tikhanski, „Open Source Load Testing Tools: Which One Should You Use?," [Online]. URL: https://www.blazemeter.com/blog/open-source-load-testing-tools-which-one-should-you-use. [Zugriff am 19 6 2016].

[30] Gatling, „Recorder," [Online]. URL: http://gatling.io/docs/2.2.2/http/recorder.html. [Zugriff am 26 7 2016].

[31] Gatling, „Advanced Tutorial: Step 01 isolate processes," [Online]. URL: http://gatling.io/docs/2.2.2/advanced_tutorial.html#step-01-isolate-processes. [Zugriff am 23 7 2016].

[32] Gatling, „Advanced Tutorial: Configure virtual users," [Online]. URL: http://gatling.io/docs/2.2.2/advanced_tutorial.html#step-02-configure-virtual-users. [Zugriff am 29 7 2016].

[33] Gatling, „Simulation Setup: Injection," [Online]. URL: http://gatling.io/docs/2.2.2/general/simulation_setup.html#injection.

[34] Gatling, „HTTP Check," [Online]. URL: http://gatling.io/docs/2.2.2/http/http_check.html#http-check . [Zugriff am 13 7 2016].

[35] Gatling, „Advanced Tutorial: Step 03 use dynamic data with feeders and checks," [Online]. URL: http://gatling.io/docs/2.2.2/advanced_tutorial.html?#step-03-use-dynamic-data-with-feeders-and-checks. [Zugriff am 17 6 2016].

[36] Gatling, „Feeder: Converting," [Online]. URL: http://gatling.io/docs/2.2.2/session/feeder.html#converting. [Zugriff am 15 7 2016].

[37] Gatling, „Feeder: non-shared-dat," [Online]. URL: http://gatling.io/docs/2.2.2/session/feeder.html#non-shared-data. [Zugriff am 13 7 2016].

[38] gatling, „Feeder: user dependent data," [Online]. URL: http://gatling.io/docs/2.2.2/session/feeder.html#user-dependent-data. [Zugriff am 14 7 2016].

[39] Gatling, „Assertions," [Online]. URL: http://gatling.io/docs/2.2.2/general/assertions.html. [Zugriff am 25 7 2016].

[40] Gatling, „Advanced Tutorial: Step 04 looping," [Online]. URL: http://gatling.io/docs/2.2.2/advanced_tutorial.html#step-04-looping. [Zugriff am 19 7 2016].

[41] gatling, „Scenario: Conidtional Statements," [Online]. URL: http://gatling.io/docs/2.2.2/general/scenario.html#conditional-statements. [Zugriff am 26 7 2016].

[42] Gatling, „Quickstart: Running Gatling," [Online]. URL: http://gatling.io/docs/2.2.2/quickstart.html#running-gatling. [Zugriff am 5 6 2016].

[43] Gatling, „Cookbook: Passing Parameters," [Online]. URL: http://gatling.io/docs/2.2.2/cookbook/passing_parameters.html. [Zugriff am 9 7 2016].

[44] Gatling, „Cookbook: Scaling Out," [Online]. URL: http://gatling.io/docs/2.2.2/cookbook/scaling_out.html. [Zugriff am 1 7 2016].

[45] Gatling, „Realtime Monitoring: Gatling-Configuration," [Online]. URL: http://gatling.io/docs/2.2.2/realtime_monitoring/index.html#gatling-configuration. [Zugriff am 29 7 2016].

[46] Graphite, „Tools That Work With Graphite," [Online]. URL: http://graphite.readthedocs.io/en/latest/tools.html. [Zugriff am 1 8 2016].

[47] S. Landelle, „Gatling User Group: HTTP/2," [Online]. URL: https://groups.google.com/forum/#!topic/gatling/jG8A9ux3jWw . [Zugriff am 20 7

2016].

[48] S. Landelle, „Gatling User Group: plan on supporting HTTP/2 (h2 and h2c)?," 1 8 2016. [Online]. URL: https://groups.google.com/forum/#!topic/gatling/jG8A9ux3jWw . [Zugriff am 9 8 2016].

[49] Gatling, „WebSocket: Send a Message," [Online]. URL: http://gatling.io/docs/2.2.2/http/websocket.html?#send-a-message. [Zugriff am 23 6 2016].

[50] JMeter, „Component Reference: Samplers," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#samplers. [Zugriff am 18 7 2016].

[51] JMeter, „Component Reference: Pre Processors," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#preprocessors. [Zugriff am 19 7 2016].

[52] JMeter, „Component Reference: Post Processors," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#postprocessors. [Zugriff am 19 7 2016].

[53] JMeter, „Component Reference: Logic Controller," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#logic_controllers. [Zugriff am 18 7 2016].

[54] JMeter, „Component Reference: Listeners," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#listeners. [Zugriff am 19 7 2016].

[55] JMeter, „Component Reference: Configuration Elements," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#config_elements. [Zugriff am 19 7 2016].

[56] JMeter, „Component Reference: Assertions," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#assertions.

[Zugriff am 19 7 2016].

[57] JMeter, „Component Reference: Timers," [Online]. URL: http://jmeter.apache.org/usermanual/component_reference.html#timers. [Zugriff am 19 7 2016].

[58] JMeter, „Getting Started: Non-Gui Mode (Command Line mode)," [Online]. URL: http://jmeter.apache.org/usermanual/get-started.html#non_gui. [Zugriff am 14 7 2016].

[59] D. Sztulwark, „Dear Abby BlazeMeter, How Do I Run JMeter in Non-GUI Mode?," [Online]. URL: https://www.blazemeter.com/blog/dear-abby-blazemeter-how-do-i-run-jmeter-non-gui-mode. [Zugriff am 14 7 2016].

[60] JMeter, „JMeter Distributed Testing Step-by-Step," [Online]. URL: http://jmeter.apache.org/usermanual/jmeter_distributed_testing_step_by_step.pdf.

[61] JMeter, „Remote Testing," [Online]. URL: http://jmeter.apache.org/usermanual/remote-test.html. [Zugriff am 20 7 2016].

[62] JMeter, „Generating Report Dashboard: Sample Configuration," [Online]. URL: http://jmeter.apache.org/usermanual/generating-dashboard.html#sample_configuration. [Zugriff am 13 7 2016].

[63] D. T., „How to generate Report Dashboard in Jmeter?," 24 5 2016. [Online]. URL: http://sqa.stackexchange.com/questions/18816/how-to-generate-report-dashboard-in-jmeter. [Zugriff am 13 7 2016].

[64] JMeter, „Generating Dashboard: Generating from an existing sample CSV log file," [Online]. URL: http://jmeter.apache.org/usermanual/generating-dashboard.html#report_only. [Zugriff am 14 7 2016].

[65] JMeter, „Generating Dasboard: Overview," [Online]. URL: http://jmeter.apache.org/usermanual/generating-dashboard.html#overview . [Zugriff am 14 7 2016].

[66] J. Admin, „Future Releases," 30 8 2015. [Online]. URL: https://wiki.apache.org/jmeter/FutureReleases. [Zugriff am 15 6 2016].

[67] D. Tikhanski, „WebSocket Testing With Apache JMeter," 5 3 2014. [Online]. URL: https://www.blazemeter.com/blog/websocket-testing-apache-jmeter. [Zugriff am 23 7 2016].

[68] Grinder, „Features of Grinder 3," [Online]. URL: http://grinder.sourceforge.net/g3/features.html . [Zugriff am 15 7 2016].

[69] Grinder, „Features of Grinder 3: Capabilities of The Grinder," [Online]. URL: http://grinder.sourceforge.net/g3/features.html#Capabilities+of+The+Grinder. [Zugriff am 15 7 2016].

[70] Grinder, „Getting Started: How do I start The Grinder?," [Online]. URL: http://grinder.sourceforge.net/g3/getting-started.html#howtostart. [Zugriff am 23 7 2016].

[71] Grinder, „Scripts," [Online]. URL: http://grinder.sourceforge.net/g3/scripts.html. [Zugriff am 15 7 2016].

[72] Grinder, „Script Gallery: Grinding a database with JDBC," [Online]. URL: http://grinder.sourceforge.net/g3/script-gallery.html#jdbc.py. [Zugriff am 16 7 2016].

[73] Grinder, „Coordination," [Online]. URL: http://grinder.sourceforge.net/g3/coordination.html. [Zugriff am 15 7 2016].

[74] Grinder, „Frequently Asked Questions: how do i simulate different users with The Grinder 3?," [Online]. URL: http://grinder.sourceforge.net/faq.html#simulating-users. [Zugriff am 16 7 2016].

[75] Grinder, „A Step-by-Step Script Tutorial: Sending the Request and the Statistics API," [Online]. URL: http://grinder.sourceforge.net/g3/tutorial-perks.html#Sending+the+Request+and+the+Statistics+API. [Zugriff am 16 7 2016].

[76] Grinder, „Script Gallery," [Online]. URL: http://grinder.sourceforge.net/g3/script-gallery.html. [Zugriff am 16 7 2016].

[77] Grinder, „Frequently Asked Questions: How do I control the number of simulated

users?," [Online]. URL: http://grinder.sourceforge.net/faq.html#threads-vs-processes . [Zugriff am 16 7 2016].

[78] Grinder, „Weighted Distribution Of Tests," [Online]. URL: http://grinder.sourceforge.net/g3/tutorial-weight-distribution.html. [Zugriff am 23 7 2016].

[79] Grinder, „The Console Service: Running without a GUI," [Online]. URL: http://grinder.sourceforge.net/g3/console-service.html#headless. [Zugriff am 17 7 2016].

[80] Grinder, „The Grinder 3: Create a script and a property file," [Online]. URL: http://grinder.sourceforge.net/g3/manual.html#createscript. [Zugriff am 24 7 2016].

[81] Grinder, „The Console User Interface: The Graphs and Results tabs," [Online]. URL: http://grinder.sourceforge.net/g3/console.html#The+Graphs+and+Results+tabs. [Zugriff am 17 7 2016].

[82] Grinder, „The Grinder3: External References," [Online]. URL: http://grinder.sourceforge.net/g3/manual.html#links.html. [Zugriff am 23 7 2016].

[83] Grinder, „The Grinder Change Log: The Grinder 3.11," 20 10 2012. [Online]. URL: http://grinder.sourceforge.net/development/changes.html. [Zugriff am 16 7 2016].

[84] G. Mulder, „Query regarding websocket support," 23 7 2015. [Online]. URL: https://sourceforge.net/p/grinder/mailman/message/34308438/. [Zugriff am 17 7 2016].

[85] Gatling, „Quickstart," [Online]. URL: http://gatling.io/docs/2.2.2/quickstart.html#quickstart. [Zugriff am 20 6 2016].

[86] Gatling, „Simulation Setup: Injection," [Online]. URL: http://gatling.io/docs/2.2.2/general/simulation_setup.html#injection. [Zugriff am 14 7 2016].

[87] Loader.io, „Plans & Pricing," [Online]. URL: https://loader.io/pricing. [Zugriff am 28 7 2016].

[88] I. Molyneaux, The Art of Application Performance Testing, 2nd Edition, O'Reilly Media, 2014, p. 278.

[89] Oracle, „Planning for Load Testing: Goals of Scalability Testing," [Online]. URL: https://docs.oracle.com/cd/E25294_01/doc.920/e15484/oltchap2.htm#autoId0. [Zugriff am 23 6 2016].